Components for office systems
Page Numbers: Yes X: 527 Y: 10.5"

Inter-Office Memorandum

To Heading: CSL

Date March 9, 1978

From B. W. Lampson

Location Palo Alto

Subject Components for office systems

Organization CSL

XEROX

Filed on: <Lampson>components.memo

Preliminaries

This memo is a discussion of the basic components which I think we will need in order to sensibly build prototype office systems in the future. It is organized in two major parts: a discussion of general issues, motivation etc., and a list of components we might consider building.

Since this is just a first pass, details should not be taken too seriously. No claim to completeness is made.

Introduction

Examples of prototype office systems which we have built in the past are Bravo, Officetalk and MC. Each of these systems was constructed pretty much from the ground up; that is, all the facilities except for the basic disk and keyboard handling provided by the operating system were constructed from scratch. This was done for two major reasons:

1) Understanding of what facilities are needed was poor, and hence it didn't seem reasonable to make the substantial effort required to design and implement a clean interface which could be reused.

2) Performance requirements and resource constraints dictated a very specialized interface and a very specialized implementation. There are two subcases here, of which I think the first has been most important:

a) The *environment* was highly specialized, e.g. in its memory management, handling of the display, or whatever, and the facilities depended heavily on the details of this specialized environment. For example, screen-painting routines must know about fonts, which in turn are large enough that they must be swapped. Thus, the screen painter becomes heavily dependent on the memory management facilities.

b) The component itself was specialized. For example, the MC editor has no formatting capabilities, at least partly because including all that machinery would probably overload the Alto.

We now have a much better understanding of what facilities are needed in a number of areas, from our experience with a fair number of systems. Furthermore, with D-0s and Dorados coming, we will have a much less constrained computing environment, and need not be so concerned about time and space. This doesn't mean, of course, that we can ignore

performance, but simply that we have quite a bit more room for maneuver.

Furthermore, I claim that we have no realistic alternative to a large amount of standardization. The amount of effort required to produce a decent component for maintaining a data base, editing a formatted document, painting the screen with arbitrary characters or splines, keeping a history list and replay file, making a Press file, or a number of other things discussed in more detail below, is so large that such components cannot be produced as side efforts in a project whose main goals are elsewhere. As a result, what tends to happen is that in most of the systems we build, these jobs are either not done, or done feebly and often sloppily, just to make it possible to get on with the main task. Furthermore, since some of the jobs *have* to be done (e.g., MC had to have an editor) effort is diverted from the goal of the research (handling messages) to a basically irrelevant goal (writing an editor). By the way, none of this is to be interpreted at a criticism of any of these systems or components; I think everyone did the right thing, in the context he was working in. I also think, however, that we must improve the context drastically, or we will get hopelessly bogged down.

Dangers

The greatest danger I see if we pursue this path is that of *excessive grandiosity and perfectionism.* We now know how to do a number of things quite well; hence we should set our sights for these components fairly high. On the other hand, if we aim too high, the components will be too large and clumsy, and will take too long to design (especially) and build. On the whole, I think it is better to err on the low side at first, and upgrade later. A reasonably clean basic design will usually support a surprising amount of upgrading. Let's not aim for perfection.

A secondary danger is *religion*, the rock on which the Humus effort foundered. For those who don't remember, I might mention that Bob Sproull originally proposed Humus with very much the goals outlined above. Unfortunately, attention quickly shifted to the basic programing environment (interfaces, allocation etc), and thence to programming language design. The resulting design effort was expended mainly on religious issues of questionable practical importance, and the implementation was so large and clumsy that it has received little use. The original goals were completely lost sight of.

Machines

The rest of this memo is a list of components which I think we should plan to build over the next year or two. I think that they should be designed on the assumption that systems built from them will *run on a D-0 or Dorado*, with at least 192k of memory (and that could be upped to 288k if necessary; D-0 memory normally comes in 96k chunks). Most of the *development* can be done perfectly well on the Alto, provided we bear in mind that complete systems embodying these components will run only clumsily, if at all, on the Alto.

Language

I have assumed without proof that *the language of implementation will be Mesa*, because I don't think we can afford to ignore either the performance advantage it has over Lisp or Smalltalk, or the large amount of effort which has been and continues to be invested in the Mesa system, its supportting environment, and components written in it.

Environment

Finished systems will run under Pilot; this seems pretty non-controversial.

Less clear is the extent to which we should try to make use of higher-level software being constructed in SDD. A lot of work has been done in the Diamond project on text images, screens, editable documents and several other things. It would be worth a lot to be able to capitalize on this work. I think a serious effort is needed to understand the strengths and limitations of the Diamond code before we can decide whether to use it or not. (In this connection, I think that the fact that it doesn't currently run under the standard Mesa runtime is not very important. The effects on the Diamond code itself are not great - the main effect is performance on the Alto - and furthermore a Diamond running under Pilot is planned.)

Style

My current feeling is that components should be cast into object form, and the enumeration below is in that style. An object is characterized by the operations defined on it, and specifying these operations would be the first step in making any of the suggestions below more precise.

Components

The list in this section is organized under a few major headings: information, media, records and services. These are only intended to be suggestive; please do not take the classification too seriously. I have arbitrarily (and perhaps incorrectly) omitted things which I have assumed to be in the environment, e.g. processes. The headings in italics try to suggest the kind of object the component knows how to deal with. They were chosen to be short and catchy -- the following text says what they are supposed to mean, and it might not be what you would guess from the heading.

A reminder: the idea behind these components is that from a set of them, plus some new code (perhaps a lot), one can construct a system which does some specific function for a user (SDD calls it an *application*). The components themselves do not do anything for a user, but instead are intended to be as general-purpose as possible. Of course components might use other components; I have not tried to indicate such dependencies here.

Information

Text image

This component knows about fonts (at least the character sizes, kerning etc), character looks, and paragraph formatting. From some encoded representation of text it can constructed formatted lines, with all the information needed to allow the line to be displayed or printed. It knows the geometry of the characters (though not their shapes) and can also translate coordinates within the formatted line back into character pointers; this function is needed for selection. It knows about styles for paragraphs (unless we find a way to separate out the style mechanism). It should be able to deal with tables and with formulas.

Subcomponents are needed which know how to paint text on the screen (including proper clipping) or on a high-resolution bitmap. The latter is needed even outside of a printer server, so that text can be properly coupled into the manipulation of high-resolution images.

Line image

These images consist of straight and curved lines, and perhaps filled areas. Obviously more than one representation is possible. Geometric transformations, variations in brushes etc. must be dealt with. Some specialized input facilities may also be appropriate. Perhaps this component should know something about how to structure a line image (connection/control points, grouping of lines into figures etc).

Again, subcomponents are needed which know how to paint the images at various resolutions. A more complex interface with the screen may also be needed to handle invalidation efficiently.

Scanned image

These images are bitmaps, again with a variety of psossible representations: compressed, uncompressed, single- or multi-bit per point, etc. We don't have much experience in this area as yet, so I won't elaborate.

Form

This is the basic mechanism for structuring various kinds of information. A form has both logical and geometric properties. That is, it has fields with names, and perhaps with a structure of nesting, repetition or dependency. Also, it has a layout, which determines how to arrange the fields on the page, and what kind of background to provide. The logical structure allows information to be extracted from the form for various purposes, e.g., to use as a summary, or as inputs to some computation, or to index the form in some data base.

This is another area where we don't have much experience, but this idea is so powerful that we certainly can't ignore it.

Audio

This component knows how to input, output, and represent audio information. It might also have some ability to generate audio from coded information, whether tones or words. It should have some idea about the structure of the audio, at least in terms of elapsed time, and perhaps in term of word breaks or whatever. Closely associated is the ability to deal with the telephone system: make and receive calls.

Word recognition is something which might be nice, but it probably requires some hardware support, and I see it as a nicety for the user interface, probably not worth the effort required to attempt it from CSL's point of view.

Media

Screen

This is the component which knows about windows, scrolling, efficient repainting of the display. Actually, very little of this has directly to do with the display screen, and all the mechanism of windows, projectors etc works just as well for any kind of imaging. On the other hand, there is stuff for deciding which window is on top, positioning windows on the screen etc which is mainly of interest in the interactive situation. I am not sure how to split this up, but we certainly have quite a few models: Smalltalk, Officetalk, Dlisp, Bravo etc.

"Menu"

As you can see from the quotes, I am not happy with this label. The idea is that some standard ways of interacting with the user are common enough to be worth codifying. Menus and selections are two obvious examples. I certainly don't think that this kind of codified interface will be as good as one more carefully designed for a particular application. On the other hand, careful design of user interfaces is an awful lot of work, and many systems will be much better off concentrating on the capabilities they are providing, and getting by with a minimally acceptable user interface.

Press file

This is our standard interface to hard copy, and it is essential that things be organized in such a way that displayable images are automatically printable. I am not a subscriber to the view that Press files should be edited, however, so I am simply talking here about a component for creating Press files. It should have the same interface as the basic screen painter.

Records

Editable document

This very important component provides the basic framework for carrying the representations of the various kinds of information we deal with. The label I have used reflects my view that the most important design criterion is being able to edit (and display) the contents of the document (after the basic requirement that all forms of information be representable). Another important consideration is the ability to handle very large documents in a reasonable way. Implementation note: there should *not* be a one-to-one correspondence between document and (operating system) file.

History list

It seems to me that the basic mechanism for capturing the sequence of transactions (changes to stored data) in an interactive session can readily be abstracted from all the details of the interaction. Such a captured sequence (history list) can be replayed, saved for analysis, used to report a bug, or used for undoing. In all cases, the structure is the same: a description of the environment (files mostly), and a sequence of transaction records of the form <operation, data for doing the operation, data for undoing the operation, comment>. The comment might be used, among other things, to describe the history to the user. It is necessary to force the records out onto stable storage at appropriate intervals.

This kind of record can be kept at various levels. In Bravo, for instance, we might keep every user interaction (e.g. every selection) or only the minimum needed to reflect changes to the document (replace and look operations). The different degrees of detail have different uses. For bug reporting, for example, any user action which causes a significant state change must be logged, even if the state change is irrelevant to the user.

This component obviously does not incorporate any such decisions. Instead, it provides the record-keeping mechanism, and defines the interface into which which clients must cast their actions. The resulting discipline on the clients is its most important contribution.

Small data base

Many of the systems we would like to build depend a great deal on some kind of data base. Examples:

> Any kind of filing-system has a data base of the filed documents, their authors, creation dates, etc.
>
> A message system has a data base of received messages, and probably other data bases of sent messages, distribution lists, etc.
>
> A calendar and scheduling system has a data base of events, subject, participants, time, priority, etc.
>
> Any accounting system has a data base, perhaps of the classical EDP kind.

Currently we get by with the operating system's directory mechanism and an occasional special-purpose mechanism like MC's table of contents. Some attempt to generalize and unify seems worthwhile (essential?).

It seems to me that a personal data base (and most of the examples above are of personal data bases) has several characteristics which make it different from the classical large, centralized, multi-access data base on which attention has focused in the past.

a) It is small, probably less than 1000 items of one or two hundred bytes each.

b) It is comparatively unstructured, i.e., quenies on a number of different keys are likely.

c) The presence of the display means that a typical retrieval is likely to involve 30 items, rather than one, leaving the user's eye to perform the final selection process.

d) The other essential operation is sorting of a few hundred items.

e) Sharing is not an issue.

These observations suggest that a specialized implementation is worth considering. The following idea, which exploits the characteristics of a disk attached to a personal computer, has occurred to me:

> Have a relational data base, with each relation stored in contiguous disk pages.
>
> Since the Alto can read data from its disk at 1 Mbit/sec and do some processing on it, a relation with 1000 entries of 120 bytes each can be scanned in one second. The D-0 or Dorado disks are 4-8 times as fast.
>
> By moving all strings of any length into a "print-name relation" and referring to them elsewhere by atom number, the size of a typical entry can probably be held to much less than 120 bytes. Also, fixed-size entries would often be feasible.
>
> Updates are simple, except for rearrangement of the disk to preserve contiguity.

Message

Messages must be created, sent, received, read and filed. Only sending and receiving (collectively called *transport*) are specific to messages, however. Since many forms of communication, both between people and between programs, can conveniently be cast into message form, an economical standard facility for sending and receiving messages seems highly desirable. Presumably it would involve the mediation of some kind of mailbox server. The cost of sending or receiving a simple message should be small, at least in the normal case. It is another question how much machinery might have to be called into play to handle exceptional conditions.

Parallel session

This is the facility provided by Telesil in one particular application -- the ability for several people to work simultaneously on the same set of data, seeing the same screen image. Like the history list, this component's main contribution is the precise specification of the interface which clients must adapt to. The implementation seems to be quite straightforward. There are some complications having to do with uniform naming of files used as input to the client applications.

Services

Dictionary

A dictionary component provides a range of servers based on the information in an English dictionary: definition of a word, hyphenation, pronunciation, spelling correction, synonym and antonym, etc.

Remote procedure call

This component has already been the subject of considerable controversy, which to do not propose to add to here. If such a thing can be provided in a cheap and generally agreeable form, I think it would be worth quite a lot. My relatively uninformed opinion is that this should be possible, if religion is not allowed to get in the way too much. Certainly some mechanism for better packaging of our communications facilities for client programs is very nearly a must.

Remote file storage

Although the larger disks available on D-0 (14 or 28 Mbytes) and Dorado (75 Mbytes) make remote file storage less important for strictly personal data, it is obviously essential for sharing. I assume that the continuing Juniper work will provide this facility for us.

Allocator

We certainly need a standard interface to one, or perhaps to two: compacting and non-compacting. The implementation is less important, since it can easily be changed. We have stumbled on this rock before, but with a lot more space available maybe we don't have to stumble again.

Cache

There is a constant need in our systems to deal with large numbers of objects: file names, messages, paragraphs, fonts, styles, etc., etc., and to take advantage of the fact that the set of frequently used objects is quite small. Perhaps the virtual memory will take care of this problem for us, but it may not, since the typical object is much smaller than a page, and hence the cost of breakage is large. A standard facility for caching such objects may therefore be needed.

Event logging

Many kinds of debugging and performance analysis are made much easier if there is a log of significant events which can be enabled with negligible perturbation of the running system. This facility has been constructed for Juniper, and a rather specialized form of it for recording Mesa transfer trap data. From these efforts we should be able to get a simple, cheap, standard method for keeping a log and examining it.

Background processing

One of the big problems with most of our systems is that they are *too slow*. Obviously faster hardware will help, but many of the most annoying delays are caused by their insistence on making the user wait while something is done whose results are not needed at once. For example, most hardcopy operations could take ten times as much real time without any inconvenience, if the machine were available immediately for other uses. Similarly for message delivery, and cleanup of the data base, in a message system. This observation seems fundamental enough that it is worth making some effort to establish a framework within which the construction of such background processes is routine. Mesa 4.0 provides the process mechanism; what we need is some conventions for scheduling, and for interlocking of data which must be shared with the foreground task.

File scan

We take very little advantage of a unique characteristic of our machines: the ability to take data from the disk at its full transfer rate, and do a significant amount of processing on the fly. This can currently be done without tremendous difficulty using the Bfs, but the method is arcane enough to be resisted by most programmers. It needs to be neatly packaged.

Why programming languages are obsolete

Terry Winograd
Stanford University

Abstract

As computer technology matures, our growing ability to create large systems is leading to basic changes in the nature of programming. Current programming language concepts are not adequate for building and maintaining systems of the complexity called for by the tasks we attempt. Just as high-level languages enabled the programmer to escape from the intricacies of a machine's order code, higher-level programming systems can provide the means to understand and manipulate complex systems and components. In order to do this, we need to shift our attention away from the detailed specification of algorithms, towards the description of the properties of the packages and objects with which we build. This paper analyzes some of the shortcomings of programming languages as they now exist, and lays out some possible directions for future research.

Introduction

Computer programming today is in a state of crisis (or, more optimistically, a state of creative ferment). There is a growing recognition that the available programming languages are not adequate for building computer systems. Of course, as any first year student of computation theory knows, they are logically sufficient. But they do not deal adequately with the problems we face in our day-to-day work of programming. We get swamped by the complexity of large systems, lost in code written by others, and mystified by the behavior of our almost debugged programs. When we look to the integrated multi-processor systems that will soon dominate computing, the situation is even worse.

This crisis in software production is far greater (in overall magnitude at least) than the situation of the early 50's that led to the development of high-level languages to relieve the burden of "coding." The problems are harder to solve, and the costs of not solving them are in the hundreds of millions. There are many ways to improve things a little, and they are being tried. But to achieve a fundamental jump in our programming capacity, we need to rethink what we are doing from the beginning.

The problem

I believe that the problem lies in an obsolete fundamental view of programming and programming languages. A widely accepted view can be paraphrased:

The programmer's job is to design an algorithm (or a computation) for carrying out a desired task, and to write it down as a complete and precise set of instructions for a computer to follow. High level programming languages simplify the writing of these instructions by providing basic building blocks for stating instructions (both control and data structures) that are at a higher level of the logical structure of the algorithm than those of the basic machine.

This view has guided the development of many programming languages and systems. It served well in the early days of computing, but in today's computational environment, it is misleading and stultifying. It focusses attention on the wrong issues, and gives the most important aspects of programming a second-class status. It is obsolete in the same sense that binary arithmetic is obsolete -- the things it deals with are a necessary part of computing, but should play a subsidiary rather than central role in our understanding of what is relevant.

As computer technology (for both software and hardware) matures, our growing ability to create complex systems has led to two basic changes in the nature of programming. Each of them has been hindered by the traditional idea of programming languages.

1. *The building blocks out of which systems are built are not at the level of programming language constructs. They are complex "subsystems" or "packages," each of which is an integrated collection of data structures, programs, and protocols.*

By making use of existing modules, a programmer can deal with design at a much higher level, creating an integrated system with capacities far greater than a program that could be built with the same effort "from scratch." Components for general tasks (such as memory management, user interface, file management and network communication) can be designed once, rather than reconstructed for each system that needs the capability. Unfortunately, in current programming practice this is more of an ideal than a reality. The difficulties of using existing packages often make it easier to replicate their function than to integrate them into a system. The only way such packages are generally used is by building programs within an "operating system" that provides facilities within a uniform environment. Only those packages that are needed by the majority of users find their way into operating systems, and the facilities for using them are complex and ad hoc relative to modern programming languages.

2. *The main activity of programming is not in the origination of new independent programs, but in the integration, modification, and explanation of existing ones.*

The second change grows from the first. As we are able to build more complex programs, we develop systems that grow to fit an environment of needs, rather than carrying out a single well-specified task. A complex system (such as one for airline reservations, text preparation and formatting, or statistical analysis) evolves over many years, increasingly coming to fit the needs of those who use it, and adding new capacities as hardware advances make them practical. As additional needs and possibilities arise, it should be far easier to modify and combine existing well-tested systems than to build new ones. In most cases, the needs for continuity in the use of the system (including "upward compatibility" for existing data and user programs) make it impractical to start from scratch. Using current programming techniques, systems often reach a point where the accretion of changes makes their structure so baroque and opaque that further changes are impossible, and the performance of the system is irreversibly degraded. The situation is further complicated by the fact that modifications are often done not by the original builders, but by new programmers with an incomplete or even wrong understanding of the system.

The difficulties in building and modifying large systems have long been recognized and lamented. They have led to various schools of "structured programming," and to the emphasis on restriction and discipline in the design and use of programming languages. There is a large body of lore shared by practicing programmers, providing ways to recognize the problems and guidelines for avoiding the most obvious of them. These include bodies of standards and conventions designed to avoid misunderstanding and miscommunication. But ultimately the solution lies not in greater discipline but in more adequate tools.

Towards a solution

Just as high-level languages enabled the programmer to escape from the intricacies of a machine's order code, *higher-level programming systems* can provide the means to understand and manipulate complex systems and components. In order to do this, we need to shift our attention away from the detailed specification of algorithms, towards the description of the properties of the packages and objects with which we build. A new generation of programming tools will be based on the philosophy that what we say in a programming system should be primarily *declarative,* not *imperative*:

> *The fundamental use of a programming system is not to provide a set of* ***instructions*** *for accomplishing a task (or carrying out an algorithm), but is to provide for the expression and manipulation of* ***descriptions*** *of computational processes and the objects on which they are carried out.*

Current languages provide only scattered specialized mechanisms for description. Declarations are a ubiquitous form of low-level description, and assertions about the state of a computation have been proposed as a basis for automatic programming and verification. But if we look at what a programmer would say about a program to a colleague who wanted to work on it or use it, very little of the description appears anywhere in the "code." If (either because of idealism or coercion) the programmer has included comments, they can provide useful, but local, description. If further (almost always through coercion) the program has been documented, there may be more global descriptions. In large systems, documentation will include a careful specification of protocols and conventions not belonging to any one program, but vital to the system as a whole. These various pieces of description are scattered, and for the most part not usable except as English text to be read. There is an increasing amount of work on "specification languages" and "structured design" formalisms, which goes in the right direction, but these still play only a peripheral role in the normal activity of programming and debugging.

> *I want to turn the situation on its head. The main goal of a programming system should be to provide a uniform framework for the information that now appears (if at all) in the declarations, assertions, and documentation. The detailed specification of executable instructions is a secondary activity, and the language should not be distorted to emphasize it. The system should provide a set of tools for generating, manipulating, and integrating descriptions. The activity that we think of as "writing a program" is only one part of the overall activity that the system must support, and emphasis should be given to understanding, rather than creating programs.*

The rest of this paper explores some of the consequences of this view, and makes some suggestions as to what a higher-level programming system might look like. It is an attempt to lay out the problems, not to solve them. It will take many years of research before these speculations can be backed up with concrete evidence.

A motivating example

One of the best ways to understand a general view of computing is to look at the examples used by those who hold it. Knuth's opus, *The Art of Computer Programming* [50], begins by discussing the Euclidean algorithm. In a clear and simple way it exemplifies the basic notions of algorithm and program from his viewpoint. The LISP 1.5 Manual [51] includes the LISP interpreter written in LISP, illustrating the manipulation of symbolic structures, and the ability to treat the programs themselves as symbolic data. The view proposed in this paper is best illustrated by an example at a very different level.

Imagine that you have come to work as a system designer and programmer for a large university. The task you are given is:

The current situation: *The university administration has a computer system for scheduling and planning room usage. Users at several sites on campus access the system through interactive graphics terminals that display building floorplans as well as text and other graphic data. There is a mini-computer running each cluster of terminals, connected to the central campus facility by a communication network. Each cluster has a device capable of printing out floorplans and graphic data like that displayed on the terminals. Large-scale data storage is at the central campus computer facility.*

The system keeps track of the scheduled usage of all rooms, including long-term lab and office assignments, regular class schedules, and special events. It is able to answer questions about current scheduled usage and availability. Querying is done from the terminals, through structured graphic interaction (menus, standardized forms, pointing, etc.) and a limited natural language interface. The system does not make complex abstract deductions, but can combine information in the data base to answer questions like "Is there a conference room for 40 people with a projection screen available near the education building from 3 to 5 on the 27th?". Users with appropriate authorization enter new information, including the scheduling of room use and changes to the facilities (including the interactive drawing of new or modified floorplans). In addition to the current assignments, the system keeps a history of usage for analysis. Standard statistical information and data representations (such as tables, bar charts, graphs, etc.) are produced on demand for use in long-range planning.

Your assignment: *The dean wants the system to provide more help in making up the quarterly classroom assignments. She wants to give it a description of the courses scheduled for a future quarter and have it generate a proposed room assignment for all courses. In deciding on assignments, the system should consider factors such as expected enrollment (using past data and whatever new information is available on estimated enrollments), the proximity of rooms to the departments and teachers involved, the preference for keeping the same location over time, and the nature of any special equipment needed. It should print out notices for each teacher, department, dean, and building supervisor, summarizing the relevant parts of the*

plan. Any of these people should be able to use the normal querying system to find out more about the plan, including the motivations for specific decisions. Properly authorized representatives of the dean's office should be able to request changes in the plan through an interactive dialog with the system, in which alternatives can be proposed and compared. When a change is made, the system should readjust whatever is necessary and produce new notifications for the people affected.

A system like this is just at the edge of our programming powers today. It would take many programmer-years of effort to build, and would be successful only if the project were managed extraordinarily well, even by the standards of the most advanced programming laboratories. But it is not hard because of the intrinsic difficulty of the tasks the system must carry out. It combines hardware and software facilities that have been demonstrated in various combinations a number of times. Even the question-answering and assignment-proposing components are within the bounds of techniques now considered standard in artificial intelligence.

The problem is in our power to organize systems. The integration of all of the components of the "initial system" would be a major achievement, calling on our best design tools and methodologies. The idea that a new programmer could come in to such a system and make changes widespread enough to handle the "assignment" is enough to make an experienced programmer shudder. It would be hard enough to add new types of questions (such as explanations for decisions), new information (such as distances and estimated enrollments), and new output forms (such as schedule summaries for departments). But even more, we are trying to integrate a new kind of data (projected plans) into a system that was originally built to handle only a single current set of room assignments and a record of their history. These projected plans must be integrated well enough for all of the existing facilities (including floorplan drawing, question answering, statistics gathering, etc.) to operate on them just as they do with the initial data base.

In order for any of this to be feasible, we need a uniform language across all of the different components and data structures, in which the objects and processes of interest can be described. This description has to be at a conceptual level that makes it possible to ignore details of implementation whenever possible, and to state the interconnections between objects and processes of widely varying structure.

Three domains of description

A system of this complexity can be viewed in each of three different "domains," *subject, interaction,* and *implementation.* Each viewpoint is appropriate (and necessary) for understanding some aspects of the system and inappropriate for others. We will look at the example from each of these viewpoints in turn, then discuss how they might be embodied in a programming system.

The subject domain. This system, like every practical system, is about some subject. There is a world of rooms and classes, times and schedules, that exists completely apart from the computer system that talks about them. The room or the course can't be in the computer -- only a description of it. One of the primary tasks in programming is to develop a set of descriptions that are adequate for talking about the objects being represented. There are descriptions for things we think of as objects (e.g., buildings, rooms, courses, departments) and also for processes (e.g., the scheduling of events). These descriptions are relative to the

goals of the system as a whole, and embody a basic view of the problem. For example, what it takes to represent a room would be fundamentally different for this system and for a system used by contractors in building construction.

All too often, the development of descriptions in this domain is confused with the specification of data structures (which are in the domain of implementation). In deciding whether we want a course to be associated with a single teacher, or to leave open the potential representation of team-teaching, we are not making a data structure decision. The association of a teacher (or teachers) with a course may be represented in many different data structures in many different components of the system. One of the most common problems in integrating systems is that the components are based on different decisions in the subject domain, and therefore there is no effective way to translate the data structures. A programming system needs to provide a powerful set of mechanisms for building up and maintaining "world views" -- coherent sets of description structures in the subject domain that are independent of any implementation. Each component can then implement part or all of this in a way that will be consistent with both the structure of that component and the assumptions made in other components.

A complex system like that described above will need hundreds of different categories of objects. Some of these (such as classrooms and courses) will be unique to the system. Others (such as times and dates, schedules, physical layouts) will be shared across a wide range of systems. They will be related into hierarchies of abstraction. We can think of a *seminar*, a *course*, and a *special lecture* as examples of a more general class of *event*, all having a time, a place, etc. For some purposes, this is the right level of generality. For other purposes, we need to distinguish carefully and use special information associated with each. A major part of building up systems will be the systematic development of descriptions that provide a uniform medium for describing components and their interactions.

The domain of interaction. This system, like every functioning system, can be viewed as carrying on an interaction with its environment. As we will discuss in a moment, the choice of "system" and "environment" is relative to a specific viewpoint, but for the moment let us consider the system as viewed by the users. In this domain, the relevant objects are those that take part in the system's interactions: users, files, questions and answers, forms, maps, statistical summaries and notifications to departments. The processes to be described are those like querying the system, describing a new event to be scheduled, and proposing a schedule for a new quarter.

The domain of interaction is concerned with descriptions that are largely orthogonal to those in the subject domain. We can talk about a question as having certain characteristics (e.g., looking for a yes-no answer) independently of whether it is about a room or a lecture. We can talk about the filling out of a form without reference to its specific contents. It is also (and more importantly) independent of the domain of implementation. From the traditional viewpoint, this independence is a bit more difficult to see than the independence of interaction and subject matter. Whereas a subject domain object (like classroom) clearly cuts across large parts of the system, an interaction object (like a question or the process of filling out a form) is typically handled by a single component and described in terms of its implementation. But for the same reasons we want to keep subject domain descriptions independent of their implementations, we must do the same with interaction descriptions.

This view can be applied recursively to sub-parts and components of the system. In looking at one part (such as the on-site processor and its cluster of terminals) we can view it as an independent system interacting with an environment including both the users and the other parts of the overall system, such as the centralized data store. Even within a single implementation module (e.g., the question-answerer), we often want to describe what is happening as an interaction between several conceptual sub-systems (the parser, the semantic analyzer, etc.). As with the larger system, it is vital to keep in mind the distinction between the interaction domain and the implementation domain. In order for two conceptual subsystems to be described as interacting, they need not be implemented on physically different machines, or even in different pieces of the code. In general, any one viewpoint of a component includes a specification of a boundary. Behavior across the boundary is seen in the domain of interactions, and behavior within the boundary is in the domain of implementation. That implementation can in turn be viewed as interaction between sub-components.

It is in the domain of interaction where there is currently the most to be gained from developing bodies of descriptive structures to be shared by system builders. There are already many pieces that can be incorporated, including protocols (e.g., network communication protocols, graphics representation conventions), standardized interaction facilities (e.g., ASKUSER and DLISP in Interlisp [52, 53], the Smalltalk display programs [31]), front-end query packages (in various artificial intelligence programs), data-base standards, and so forth. Currently each of these is in a world and formalism of its own. Given a sufficiently flexible tool for describing and integrating interaction packages, this level of description will be one of the basic building blocks for all systems.

The domain of implementation. Every computer system operates on a set of physical devices with hard-wired mechanisms for for storing and manipulating data. It is in this domain that we normally think of programming. The detailed choice of algorithms, data structures, and configuration is determined by properties of the hardware and of the descriptive languages we have available for specifying its behavior. However, it would be a mistake to equate this domain with our current notions of programming. The objects in this domain include everything from individual memory bits and subroutines to subsystems (e.g., the file system, the memory management system, the operating system), running processes, hardware devices, and code segments. They include those things we talk about in programs, those we talk about in the debugger, and those found in the machine and hardware manuals. In this domain, as in the others, a uniform system for description is needed, which is not primarily a language for specifying a set of instructions.

In addition to all those things that are directly derivable from the code, the manuals, and the state of the machine, there is also a body of process description. For example, it may be a property of a specific memory-management package that it periodically undergoes a garbage collection period of up to 5 seconds during which no new memory allocations can be made. Such "performance" characteristics may be vital for understanding the interactions of a component with the rest of the system, but are not explicit in its code. Other descriptions bring into focus things that may be implicit in the code. A file system may delete a file if its creation process is interrupted in certain ways that would leave it in danger of being inconsistent. The code that does this may be distributed through various checks and actions, but for the programmer attempting to understand the program it is necessary to have a coherent overview of what is happening.

Similarly, many of the things we think of as properties of data structures are actually conventions spread through the code that manipulates them. Much of the work in structured programming has been to isolate these conventions into access functions that go into a "module" with the data structures [29, 32, 34]. The object-oriented style of programming encouraged by Smalltalk [31] is another attempt to provide this kind of modularity. A system for implementation description would provide for stating these in a more general way along with those things that now appear only in the comments. As with procedures, data structures can also have implicit properties (e.g. the expected maximal size of variable length fields) that need to be stated explicitly in order for a person to understand how they will interact with other components.

The boundary between hardware and software has been blurred in the past few years through developments such as micro-code, uniform logic arrays, and the extensive use of virtual machine architecture. A programming system based on description pushes this one step further. The emphasis is on an overall description of a component rather than the instructions needed to cause some piece of hardware to run it. A piece of software and a piece of hardware designed to achieve the same purpose would have descriptions that differed in details (e.g. timing), and in the specific aspect that described the code (or logic circuits) used to carry out the steps. They might be identical in the domain of interaction, and even to a large degree in the domain of implementation (for example in the logical description of their data structures).

A sketch of a higher level programming system

So far, we have been talking in a general way about the different domains of description and the kinds of things that might be said about a component or system from each of their viewpoints. This doesn't yet provide a coherent picture of what a program will be. What do we see on the printed page or on the screen? How is it organized? How do we do anything with it?

Once again, it is important not to let our preconceptions get in the way. Notions such as code, listing, file, and compilation are based on the idea of a program as a set of instructions. There need not be any directly corresponding objects in a higher-level system. Instead, it should be based on something much more like what we now think of as an artificial intelligence system, with its "knowledge base" of assertions and procedures. There will be a set of interrelated descriptions, stored in a form that makes it possible to retrieve, manipulate and display them. These will include *prototypes* for categories of objects and processes (like *classroom*, and *filling out a form*), and *instances*, which correspond to individual objects and processes in one of the domains (such as the course CS 365 in Winter 1978, the contents of the third page of next quarter's schedule, and the process currently running in the data-base server). Instances can be described by more than one prototype, and prototypes are related into hierarchies with different degrees of abstraction. The details of all this are still a matter for extensive research. One set of possibilities is being explored in KRL [2,3], but the basic idea of higher level systems could be implemented using other descriptive representations, such as semantic networks [4, 6, 10, 11, 12] and other frame based systems [5, 7].

In addition to the basic description system, there will be a wide range of commonly useful prototypes and instances. Some of them (e.g., graphics formats, dates and times, communication protocols) will be in the subject and interaction domains. Others (such as the data structures used in a particular data base) will be in the implementation domain. Some (such as descriptions of specific pieces of hardware and software that are being used) will be specific to the programming system. Others (such as those for abstract objects like sets and sequences, and for process structures) will be very general. In approaching a problem, a programmer will make use of this vocabulary of concepts and descriptive categories, both for interfacing with existing components and organizing new ones.

A programmer's use of such a system will be highly interactive. Since the understanding of a component comes from having multiple viewpoints, no single organization of the information on a printed page will be adequate. The programmer needs to be able to dynamically reorganize the information, looking from one view and then another, going from great generality down to specific detail, and maneuvering around in the space of descriptions to view the interconnections. This will require an interface that is more sophisticated than the question-answering interfaces now used on artificial intelligence knowledge bases. It seems likely that pictorial representations, interactive graphics, and "intelligent summarizing" will play a large role.

Of course, there always remains the task of providing a description of each component that is detailed enough to allow the system to run it. This will be part of providing a broader description, and may be done in stages. A very abstract specification of what a component does will be sufficient for a kind of "high-level debugging" in which its interactions with other components can be analyzed and described without carrying out the steps at the the lowest level. There is a whole range of operations that are now thought of as "automatic programming", which will enable the programmer to let the system fill in details once the overall behavior of the component has been specified. Some of this will be based on standardized defaults, others on automated analysis of performance characteristics. It will all be based on the availability of descriptions in the implementation domain of the various machines and subsystems being used.

As systems become more complex, the level of desirable invisibility will rise. Current high-level programming languages do not give the user the opportunity to decide just how the hardware registers of the machine will be used to store variables, since there is much more to be gained by a uniform integrated approach to their use by the compiler. Similarly, much of what we now think of as data structure and algorithm specification will be handled by programs that can take into account much more complex efficiency considerations than are practical for a human programmer.

In summary, a programmer will use a programming system that contains a base of knowledge about the system he or she is working on, and of other potential components and concepts of programming that might be of use. The programmer will modify the descriptions of existing components, and create high-level descriptions of new ones to be added. These modifications and additions may be from the viewpoints of all of the different domains, and will be done in cooperation with the system, which checks for consistency, looks for consequences of new information, etc. Checking and debugging will be done at a variety of levels as the description becomes more detailed. The system will attempt to fill in details that are needed to completely specify the implementation, so the system as a whole can be run. The debugging process will make use of sophisticated reasoning processes that

can use all of the different domains of description in analyzing and reporting what is happening.

What needs to be done

To create a system like the one just described, we need further research in three distinct but interrelated areas: the development of an effective descriptive calculus; the creation of a body of descriptive concepts for computational processes; and the building of a complex integrated system which uses them.

A descriptive calculus. The main thrust of these ideas depends on the ability to create and manipulate descriptions in an effective, understandable way. There are existing formalisms for description (for example, the predicate calculus) which are clear and well-understood, but lack the richness which makes natural description effective. They can be used as a universal basis for description but only in the same sense that a Turing machine can express any computation. They lack the higher-level structuring which makes it possible to manipulate descriptions at an appropriate level of conceptual detail.

The requirements for a descriptive language of the kind I propose are quite different from those used in the mathematical foundations of computation, or for program verification. Work in these areas (see, for example the collection of papers in [15]) emphasizes the use of descriptive languages in rigorous proofs of the properties of programs. A higher-level programming system must instead emphasize the use of descriptive languages for communication. The concentration must be on those aspects which aid in giving a person a clear overall understanding (at variable levels of detail and from multiple points of view), rather than on those aspects which increase the mathematical tractability of the descriptions. There are artificial intelligence formalisms (such as semantic networks [4, 6, 10, 11, 12] and KRL [2,3]) with increased dimensions of expressiveness, but are not yet at a level of precision which would make them sufficiently understandable to be used in a system of the required complexity. The characteristics which they explore (and which will need to be part of a formalism to be used in a higher-level programming system) include:

Prototype hierarchies. The nouns and verbs of a natural language can be organized into hierarchies (or taxonomies) which capture much of the logical structure of what they describe. We know that a *dog* is an *animal*, and the answers to questions about dogs will often be derived through general properties of animals. Systems such as semantic networks treat these deductions specially, rather than dealing with them uniformly as a set of universally quantified implications. This leads to greater efficiency for the common calculations, and provides a structure which makes it much easier for a programmer to organize a knowledge base. These hierarchies contain information which could be thought of as a set of independent facts, but has additional structure in the same sense that a structured program structures a set of steps and jumps.

The centrality of defaults. Most logical calculi are optimized for handling generalizations which are either true or false. They do not provide means for stating generalizations that are not completely universal, but are "usual" or "normal" or "expected". In natural descriptions of any kind, people draw heavily on the ability to use information of a less formal sort, and to use a kind of *ceteris parabus* reasoning in which a standard fact is assumed true unless there is an explicit reason to believe the contrary. One of the major directions in artificial intelligence representation research is

the attempt to provide this capability in a formally clear system. The notion of a *default* value is familiar to every programmer, but its place in a formal calculus needs to be carefully worked out.

The suppression of exceptional details. One of the major reasons for using precise formalizations is that they make everything explicit. For some purposes this is good, but there are times when understanding can come only through the suppression of detail. If we are trying to formally describe a program which has a normally simple input-output behavior (e.g. one that copies data from one place to another) we want to describe that behavior in a way which highlights that simplicity. If there are exceptional cases (e.g. when the storage allocator fails to find a sufficient block), these need to be described, but in a secondary place. The basic description of an object cannot be cluttered up with all of the details needed for handling the contingencies. Formalisms used in denotational semantics for programs demonstrates this problem well. In order to deal with a special escape at all, they demand that even the simplest programs be described as operating on continuations, environments, etc. and this description permeates every level of what is said.

Multiple levels of abstractions and instances. In dealing with programs and processes, we run into complexities involving the *instantiation* of general classes. For example, a specific algorithm (such as Euclid's algorithm) can be thought of as an instance of a general class (numerical algorithms), or as a class whose instances are programs implementing the algorithm. Each of those programs is in turn both an instance (of the class of formal objects known as programs) and a description of a class of process instances, each of which is carrying it out. If we look at the finer structure of programs, such as the instantiation of variables or pieces of code within loops, similar phenomena arise. Higher-order and typed logics deal in certain ways with the notion of a class (predicate) which is also an instance, but their austerity makes them inadequate for capturing the rich set of ways in which people interleave levels of abstraction. Artificial intelligence formalisms have not yet dealt adequately with these issues, which are currently a topic of active investigation.

A basis for describing processes. Given a formalism for descriptions in general, we need a body of prototypes for describing those things which are common to all of our programs (e.g., processes, programs, data structures, communication acts). This is a necessary kind of *library*, just as a library of standard data-structures and statistical routines is a necessary part of a system for statistical manipulations. It need not be fixed once and for all, but a good deal of it must be in place before the system is usable, and it must be held relatively uniform if the system is to be extendable.

In this area, there are many ideas floating around that need to be captured in a more precise form. The success of higher-level programming systems will depend on having a coherent body of descriptive categories which can capture all of their variety.

Modularity and structured procedures. There has been a good deal of attention in recent years to the higher-level structure of control constructs. In addition, languages based on data abstractions (such as CLU [32], Alphard [34], and Mesa [29]) provide for larger *modules* which encapsulate collections of data structures and procedures. Beginning from a different point of view, *structured system description languages* [16, 17] provide a body of conceptual tools for describing the overall structure of large

systems. We need a consistent way of talking about modularization and interaction between semi-independent modules which can be applied to system structure at all different levels of detail.

Structured data objects. Current work on programming language constructs often emphasizes the structure of the sequence of operations, in terms of loops, recursive calls, etc [14]. A related notion in describing processes is the ability to hide detail by allowing the combination of objects into a larger "structured object", and to define unitary operations on this higher level object which invoke collections of operations on the components. This has been explored for simple mathematical objects (e.g. lists in LISP's MAP functions [51], vectors and arrays in APL [20], sets in SETL [26] and VERS [19]), and seems applicable to more specialized semantic objects (in all of the three domains) as well. In many cases, much of what is now thought of as control structure can be implicit in the data structure, leading to notions of "nonprocedural" or "procedureless" languages [21, 22, 24]. The interaction between control and data structure needs to be put into a theoretical framework.

Program factoring -- objects and procedures. In viewing a process as a structured sequence of individual steps, there are different ways to think about what each of those steps is. Most programming languages lead the programmer to think in terms of *operations* (either primitive or programmer-defined) to be carried out on *arguments.* Some (such as Simula [27], Smalltalk [31] and Plasma [47]) think of typed *objects* which receive and interpret *messages.* Instead of organizing code around a single procedure (e.g. *print* or *plus*) which selects its action according to data type, they define *classes* (such as integer, list, etc.) which select what to do on the basis of the message. Artificial intelligence languages take a more general approach in using *pattern directed invocation* [1, 28]. Each step specifies a pattern (or *goal*) to be achieved. A data base of *pattern-action pairs* is accessed to decide what steps to carry out. Each of these viewpoints is best for some aspects of programming, and we need to understand how to integrate them into a larger framework.

States and transitions. There are two complementary ways of looking at a computational process -- as a sequence of operations, or as a sequence of states. This duality has been exploited in the mathematical theory of computation, but has not really been integrated into programming languages. Transition nets and Petri nets [38, 40] are state-oriented, rather than operation-oriented. Production systems [43] are state-oriented, with each production specifying a partial state description, and an appropriate transition function (not the name of the new state, but a set of operations which produce the new state). Languages which provide ways of specifying actions to be taken on special conditions [37] are really mixing state-transition description with the normal operation sequence. As with the operation/object distinction above, the goal is to find a synthesis which allows a process to be described using a mixture of the conceptual viewpoints, and to be run on the basis of that description.

Interacting processes and communication. The notions above deal primarily with a single process. The most significant direction in computing over the coming years will be towards multiple processes, both virtual (e.g. organizing a speech system as a series of separate processes, even if it runs on a single PDP-10 [46, 49]) and actual (e.g. networks of computers cooperating on a single task). There are a number of issues which have been dealt with by system designers at lower levels (like operating systems) which have

not found their way into higher level languages. There is also a wealth of metaphors provided by thinking of a computation as being carried on by a collection of independent individuals which must communicate by exchanging messages in a common language. We can draw many analogies from human communication -- What language do they talk? Which subsystems need to be multi-lingual? What are the discourse rules for establishing and controlling message flow? Is it possible to learn a second language? How can two processes make use of shared knowledge to increase the efficiency of communication? How can one process make use of an internal model of another process, in order to facilitate communication and cooperation?

A complex system. The kind of higher level programming system described here is itself a massive and complex system. Its subject matter is not an external one, like room-scheduling, but the reflective one -- the subject of programming itself. There is a bootstrapping problem. The system needed to realize these ideas can be built only with a set of tools that help in the construction of large integrated systems. Perhaps our current systems are good enough to start the bootstrapping, but that first step will be a big one. It will take a good deal of careful system-organization research before something of this scale can be effectively constructed.

Conclusion

The title of this paper is consciously provocative. Rather than asking how we might improve programming languages, I argue that we should look at things from a completely different vantage point in which there is little concern with specifying programs as we now see them. It grew out of work on description systems which emphasize concepts such as multiple viewpoints for description, default knowledge, and prototypes. These ideas can be applied to make the rich set of programming concepts represented in the literature more applicable to the real practice of programming. In order to make significant progress, we need to deal with the problems of "programming in the large". Once we begin to deal with networks of independent processors, it will become even more important to deal explicitly with global properties that cannot be understood on the basis of individual program instructions.

I believe that the the type of higher level programming system described here is a step in the right direction. There is no way to demonstrate this with certainty. As stated at the beginning, this paper is not a presentation of a worked-out solution. All we have at this point is an intuition that a particular way of approaching the problem will lead to new insights and results. The paper is intended to provoke thinking and suggest some new directions. If they turn out to be fruitful directions, we have years of hard and exciting work ahead.

Acknowledgements

In a paper of this kind, it is impossible to properly credit the sources of the ideas. It grew out of ongoing interactions with people who hold very similar ideas in different forms, and it is really just an expression of the current state of my intellectual environment. My joint work with Danny Bobrow and Brian Smith on the theoretical foundations of KRL has been a primary source of ideas, and the rest of the Stanford/Xerox KRL research group (David Levy, Mitch Model, Don Norman and Henry Thompson) have been involved in all stages of our thinking. Stanford computer science students in the CS365 seminar in 1977

pushed and probed on many of the ideas about procedures, which in turn came from the authors of the papers we read there (included in the list of references). The Xerox PARC environment has been a context in which the problems of "programming in the large" are well understood, and has provided a wealth of ideas and examples, including the work of Alan Kay and his group on Smalltalk, the implementation of the Mesa programming language, the development of programming environments by Warren Teitelman and Larry Masinter, Bob Sproull's understanding of graphics systems and protocols, and Peter Deutsch's views on system organization and programming environments. In addition, the cybernetic notions of Humberto Maturana as introduced to me by Fernando Flores have led to subtle but very important shifts of perspective in the way I look at systems of all kinds. I am also grateful to Allen Perlis, Peter Deutsch and Jim Horning for extensive and insightful comments on an earlier draft of this paper.

References

Note for the draft version: This reference list is still somewhat rough. It contains all the references from the Stanford seminar mentioned in the last section, along with others mentioned in this paper. It may be better to make it more selective, so I have not bothered to clean it up totally, but will wait for comments first. My idea in including it was that it gives a broad picture of the kinds of issues which need to be taken into account. I am a little worried that it is so broad as to be overwhelming. It is also somewhat idiosyncratic, since it consists of papers that I was familiar with. I welcome suggestions of additional papers or substitutes on all of the topics.

Description Formalisms

[1] Daniel Bobrow and Bertram Raphael, "New programming languages for AI Research", *Computing Surveys* 6:3 (September 1974).

[2] Daniel Bobrow and Terry Winograd, An Overview of KRL, a Knowledge Representation Language, *Cognitive Science* 1:1, January 1977, pp. 3-46.

[3] Daniel Bobrow, Terry Winograd, and the KRL research group, Experience with KRL-0: One cycle of a Knowledge Representation Language, *Fifth International Joint Conference on Artificial Intelligence*, pp. 223-227.

[4] Ron Brachman, What's in a concept: Structural foundations for semantic networks, *Int. J. Man-Machine Studies* (1977) 9, pp. 127-152.

[5] Randy Davis, Knowledge about representations as a basis for system construction and maintenance, in *Pattern Directed Inference Systems*, Academic Press (in press).

[6] Richard Fikes and Gary Hendrix, A network-based knowledge representation and its natural deduction system, *Fifth International Joint Conference on Artificial Intelligence*, pp. 235-246.

[7] Ira Goldstein and Bruce Roberts, Nudge, a knowledge-based scheduling program, MIT AI-MEMO 405 (February 1977).

[8] Pat Hayes, Some problems and non-problems in representation theory, AISB conference, 1974, pp. 63-79.

[9] Pat Hayes, In Defence of Logic, *Fifth International Joint Conference on Artificial Intelligence*, pp. 559-565.

[10] Hector Levesque, A procedural approach to semantic networks, TR-105 Dept. of Computer Science, U. of Toronto, 1977.

[11] David Rumelhart and Donald Norman, The Active Structural Network (Chapter 2) from *Explorations in Cognition*, 1975.

[12] Bill Woods, "What's in a link?", in Bobrow and Collins (eds.) *Representation and Understanding*, 1975.

Formalisms for specifying programs

[13] Burstall, R.M. and Goguen, J.A., Putting Specifications Together, 5IJCAI, 1977.

[14] Edsger Dijsktra, *A Discipline of Programming*, Prentice Hall, 1976.

[15] E.J. Neuhold (ed.), *Formal Description of Programming Languages*, North-Holland, 1978.

[16] Kenneth T. Orr, *Structured Systems Development*, Yourdon Press, 1978

[17] Douglas Ross, Structured Analysis Design Technique, ????

[18] R.D. Tennent, "The Denotational Semantics of Programming Languages", *CACM*, August 1976, pp. 437-453.

Structured objects and structured procedures

[19] Jay Earley, High Level Operations in Automatic Programming, SIGPLAN notices 9:4 (1974), pp. 34-42.

[20] A.D. Falkoff and K.E. Iverson, The design of APL, *IBM Journal of Research and Development*, 1973, pp. 324-334.

[21] Clair Goldsmith, The design of a procedureless programming language, SIGPLAN notices 9:4 (1974), pp. 13-24.

[22] Hammer, Howe, and Wladawsky, an Interactive Business Definition System SIGPLAN notices 9:4 (1974), pp. 25-33.

[23] Robert Kowalski, Predicate Calculus as a programming language, IFIP proceedings, 1975.

[24] Burt Leavenworth and Jean Sammet, An Overview of nonprocedural languages, SIGPLAN notices 9:4 (1974), pp. 1-12.

[25] John Reynolds, GEDANKEN--A Simple typeless language based on the principles of completeness and the reference concept, CACM 13:5 (May 1970), pp. 308-319.

[26] Jacob Schwartz, On Programming: an interim report on the SETL project; Installment I: Generalities, NYU Courant Institute, February 1973.

Program factoring -- modules, objects, and procedures

[27] Birtwistle, Dahl, Myhrhaug and Nygaard, SIMULA BEGIN, 1973.

[28] Randy Davis, Generalized procedure calling and content directed invocation, *Proc. ACM Conference on AI and Programming Languages*, August 1977.

[29] Charles M. Geschke, James H. Morris Jr., and Edwin H. Satterthwaite, Early Experience with Mesa, *CACM* 20:8 (August 1977), pp. 540-552.

[30] Chuck Geschke and Jim Mitchell, *On the problem of uniform references to data structures*, Xerox PARC CSL-75-1, January 1975.

[31] Adele Goldberg and Alan Kay, *Smalltalk-72 Instruction Manual*, Xerox PARC SSL 76-6, 1976.

[32] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert, Abstraction Mechanisms in CLU, *CACM* 20:8 (August 1977), pp. 564-576.

[33] Vaughn Pratt, The Competence/Performance dichotomy in programming, 4th ACM symposium on the principles of programming languages, 1977, pp. 194-200.

[34] Mary Shaw, William A. Wulf, and Ralph L. London, Abstraction and verification in Alphard: Defining and specifying iteration and generators, *CACM* 20:8 (August 1977), pp. 553-562.

[35] Guy Steele, LAMBDA, the ultimate imperative, MIT-AI Memo 353, March 1976.

[36] Guy Steele, LAMBDA, the ultimate declarative, MIT-AI Memo 379, November 1976.

States and transitions

[37] John Goodenough, Exception Handling: Issues and a proposed notation, CACM 18:12, December 1975, pp. 683-696.

[38] Anatol Holt, Introduction to Occurrence Systems, in Jacks (ed.) *Associative Information Techniques*, Elsevier 1971. pp. 175-203.

[39] E. Humby, *Programs from Decision Tables*, McDonald/Elsevier, 1973.

[40] P.E. Lauer and R.H. Campbell, A Description of path expressions by Petri Nets, Second ACM symposium on principles of programming languages, 1975, pp. 95-105.

[41] Howard Lee Morgan, Event Sequenced Programming, Cornell Dept. of operations research Tech report 119 (July 1970).

[42] Chuck Reiger, "The Commonsense algorithm as a Basis for Computer Models of Human Memory, Inference, Belief and Contextual Language Comprehension", Schank and Nash-Webber (eds.), *Theoretical Issues in Natural Language Processing*, 1976. pp. 180-195.

[43] Mike Rychener, Production Systems: a case for simplicity in AI Control Structures, draft of paper submitted to ACM National Conference, 1977.

[44] Earl Sacerdoti, The non-linear nature of plans, 4IJCAI, pp. 206-214.

[45] Michael Zisman, A Representation for Office Processes, Wharton Department of Decision Sciences Working paper 76-10-03.

Interacting processes and communication

[46] Jeff Barnett, Module linkage and communication in large systems, in D.R. Reddy (ed.) *Speech Recognition*, pp. 500-520.

[47] Carl Hewitt, Viewing control structures as patterns of passing messages, MIT AI Memo 410 Dec. 1976.

[48] Lampson, Mitchell and Satterthwaite, On the transfer of control between processes, Proceedings of Programming Symposium, Paris, April 1974 (#19 in Lecture notes in Computer Science, Springer Verlag, 1974), pp. 181-203.

[49] Victor Lesser, Parallel processing in speech understanding systems: A Survey of design problems, in Reddy (ed.) *Speech Recognition*, 1975, pp. 481-499.

Other references

[50] Donald Knuth, *The Art of Computer Programming*, Vol 1, *Fundamental Algorithms*, Addison Wesley, 1968.

[51] M. Levin, et. al., *The Lisp 1.5 Programmer's Manual*, MIT, 1965.

[52] Warren Teitelman, A display oriented programmer's assistant, *Fifth International Joint Conference on Artificial Intelligence*, 1977, pp. 905-915

[53] Warren Teitelman, et. al., *Interlisp Reference Manual*, Xerox PARC, 1975.

Inter-Office Memorandum

To	CSL, Smalltalk interest	Date	May 8, 1978
From	P. Deutsch	Location	Palo Alto
Subject	Why there should be no Dorado Smalltalk	Organization	CSL

XEROX

Filed on: <Deutsch>dsmallno.memo on Maxc1

This memo is an attempt to justify exactly the opposite point of view from the one I have been promoting for approximately the past 10 months. I have been taking the "positive" view that it is worth investing a lot of careful work in a Dorado Smalltalk system as the basis for a new Experimental Programming Environment. This seems to have met with a lot of polite agreement in principle, little high-level assistance in designing or putting together such a system, and no evidence that anyone wants to help grow or maintain it, let alone think about the research issues associated with a Smalltalk-based EPE as such. Thus, I am presenting the "negative" view in the hope that it will either (a) uncover a lot of people who agree with it but have been keeping quiet out of courtesy, or (b) cause people to re-examine their opinions to an extent which will lead them to embrace one or the other of the two views with some conviction.

The thesis of this memo can be summarized as follows. There are three natural user communities for a new Dorado Smalltalk, and the current approach can't be justified for any of them.

> The Understander and KBA groups won't want to use it because the semantics are less clean than those of Lisp, there is no program-manipulable representation of programs, and it lacks Interlisp's user aids.
>
> People writing new experimental systems won't want to use it because it is too inefficient compared to Mesa, and because all the packages they will want will be written in Mesa.
>
> Work on EPE in its own right, assuming anyone becomes interested, won't be done in Dorado Smalltalk because there won't be any real clients for it, and because it will be incompatible with the LRG implementations, preventing us from taking advantage of their constant stream of interesting ideas.

If one agrees with this memo, one might draw the following conclusions from it (among others):

> We should put more work into Dorado Lisp, and into an interface between Lisp and Mesa.
>
> We (other than the Understander and KBA groups) should not try to build experimental systems in anything but Mesa, possibly with a Lisp veneer.
>
> We might map over the Alto Smalltalk to the Dorado, because it is easy, but only as a "toy" system.

Smalltalk is conceptually messy compared to Lisp

The current Smalltalk message-passing semantics are cleaner than the Lisp function call semantics, but the handling of name scopes and object lifetimes is a mess, and likely to get worse in the system being designed for the Dorado. Smalltalk also has no reasonable program-manipulable representation of programs. In contrast, Scheme, a Lisp dialect developed at MIT, substantially cleans up the semantics of Lisp and also adds the ability to form closures which are functionally equivalent to Smalltalk instances (including multi-level instances which are not possible in the current Smalltalk).

Smalltalk will have to duplicate all the Interlisp user aids

If a Smalltalk and an Interlisp both exist on the Dorado, given that Smalltalk's advantages at the language level are real but modest, then Interlisp users will only switch to Smalltalk if the same tremendous volume of user aids -- Masterscope, Helpsys, display interface, some of DWIM, etc. -- is available. While re-thinking will simplify some of these areas and reduce the amount of work to be done, the sheer mass of functional capability is very large.

Smalltalk is inherently too inefficient compared to Mesa

Without adding a great deal of complexity to the compiler to provide for declarations, and to the instruction set to allow it to manipulate pre-type-checked 16-bit quantities, Smalltalk is likely to run a factor of about 3 to 5 slower than Mesa in general, and perhaps as much as a factor of 10 or 15 on inner loops. A factor of 3 to 5 in performance is more than enough to spell the difference between a usable system and an unusable one. (Lisp suffers from a similar performance penalty.)

Smalltalk will not be the language of choice for building packages

Butler's recent memo on building blocks makes it clear that we (with SDD perhaps playing a role) will be building a large number of packages in Mesa designed specifically for use in experimental systems. This suggests that the role of Smalltalk would be limited to the "glue" that holds these packages together and the "wrapper" that interfaces them to the user. A major new programming system is not needed for this purpose.

Smalltalk is a "moving target"

We are attempting to capture in the Dorado Smalltalk design as many good ideas as we can from KRL and from the ideas currently circulating in LRG about more general notions of part-whole relations (initially based on Alan Borning's work). LRG is not going stop innovating just because another implementation of some version of Smalltalk exists. Nor will CSL's Understander Group fail to discover changes in Smalltalk which would greatly enhance their ability to move forward into the next round of KRL. This means that either we commit ourselves to a lot of ongoing maintenance work trying to track LRG's innovations (and vice versa), or we resign ourselves to the certainty that within a year or so there will be very few programs that will run on more than one system (LRG is also contemplating designs for the NoteTaker).

If we want a Dorado Smalltalk, we don't need to design a new one

One perfectly feasible route to a Smalltalk on the Dorado is to take the current Alto Smalltalk, rewrite the microcode, move some of the primitives which are current performance bottlenecks from assembly language into microcode, and use the Dorado memory in excess of 64K as a cache for the disk. The principal limits on such a system would be the limit of 64K objects (16-bit virtual addresses), and the inability to communicate with other language systems. Such a system could be put together in a couple of months.

Inter-Office Memorandum

To	P. Deutsch	Date	May 9, 1978
From	T. Winograd	Location	Palo Alto
Subject	Why there might be no Dorado Smalltalk	Organization	CSL

XEROX

Filed on: <Winograd>dsmallno.memo on Ivy

Allright Peter, you finally provoked a comment!

I speak from one particular (and not terribly representative) point of view, since my main concern is with higher level description languages like KRL, and focusses more on theoretical than on performance issues. I do think, though that there are some observations with general applicability.

I cannot imagine implementing the next version of KRL in Interlisp and having any sense of at all of what is going on inside it. Surely you have tongue in cheek when you suggest that "...the semantics [of Smalltalk] are less clean than those of Lisp.." when referring to Interlisp. As an exercise in programming language semantics, describe the semantics of the construct (FOOO A:B) which might appear in an Interlisp program. Be sure to take into account LAMBDAS and NLAMBDAS (speread and nospread), DWIM and the spelling corrector, CLisp infixes and iterative operators, the record package, macros, read macros, and the effects of declarations (or lack thereof) on the interpreted, compiled, and block-compiled versions.

You mention that the handling of name scopes and object lifetimes is a mess in Smalltalk. Ask the man or woman who has programmed spaghetti and juggled a stack pointer into oblivion (again I suspect a touch of knowing irony in your critique). You state that Smalltalk has no reasonable program-manipulable representation of programs. I'm not sure how true that is, but if you have ever tried editing an expression like (NEWLIST← <X+Y 3>) in the Interlisp editor, you might be happier with plain old token vectors. In terms of all the user aids, our experience with KRL-1 has demonstrated how frustrating it can be to have all sorts of user aids (editor, file package, etc.) which almost (but not quite) work for new kinds of objects (like KRL units). Many many hours have gone into fitting square pegs into round (but user-stretchable) holes.

So where does that leave us. I claim that the next round of KRL (or something like it) will have to be built on a much cleaner and simpler base which is organized around a display-oriented personal computing environment. It will have to have lots of "packages" available, like those discussed in Butler's memo. It will only have to be relatively efficient, and the speed of making program changes will be much more important than the speed of running. What are the choices? The real issues aren't the technical ones, but the political ones -- not what can be done, but who can we count on to do it. There are several answers:

1. ASD, SDD etc.: This is the major argument for Mesa. There is a whole world out there to share with, and they are already hard at work. The conceptual shortcomings of the language will have to be programmed around (e.g. with pre-processors (but remember Humus!)) and we will have to build our own tools to suit an experimental environment, but the initial base is big and solid.

2. LRG: Count on LRG, and do the minimum needed to keep their Smalltalk

running on Dorado. This is an easy route, and again, there is a body of stuff already existing. The obvious danger is that the tradeoffs made for the benefit of small machines will become major stumbling blocks (remember the Interlisp 18 bit address space)!). In general, their taste in programming environments is likely to be much closer to your idea of an EPE than that of the development organizations.

3. MIT: You mention Scheme, and its advantages apply to its successors, such as the Lisp being done for the S-1 and VAX (by Steele, White, and Gabriel). This will be a largely machine-independent implementation, combining some of the best ideas of the Lisps of the last few years. Since it is being designed from the ground up, it will be a lot cleaner than Interlisp, and there will be a community of user/developers. However Error 33 doubles (squares?) its magnitude on crossing the Charles.

4. CSL: Well, who's left? As you have argued for months, EPE won't fly unless several people get committed to it. No one person can achieve the critical mass to make the system habitable.

There is a lot of force behind choice #1. The masses are out there, and people within CSL are rapidly gaining Mesa experience. Maybe it is possible to retrofit an appropriate experimental programming environment into the Mesa/Pilot world, including the room to make extensions (e.g. object-centered programming). I think the efficiency issue is not a valid argument for this, for two reasons. First, by the time we get appropriate extendability, the generality will reduce Mesa's theoretical efficiency. Second, for the kind of EPE you envision, efficiency of programmer-interaction dominates efficiency of code running.

Choices #2 and #3 are essentially de facto versions of choice #1 (or worse, of having no generally used programming environment at all), since the majority of the lab will continue to use Mesa, and the development of programming environments will be out of our hands. The only realistic alternative is taking #4 seriously, which you have been trying to get us to do for ages. This means convincing some subset of people (either from the Lisp end or the Mesa end) that it is worth their time. Unfortunately, as in the case of the little red hen, only an existence proof seems to have any effect.

Unfortunately, having gotten this far I realize I have no good answers. I'm not at a place where taking on a programming project is feasible (e.g. I have a book to finish, etc. etc.). My co-Lispers either have equal excuses or don't seem to share the views expressed above sufficiently to want to put effort into shifting over. I think there is some justification in our feeling that since our primary research directions are not in programming environments, that we would like to be able to look to other people in the lab to tackle them as a major research area. The problems are important enough to justify first-class status, not to be done as side-projects. Maybe that is just trying to shift the burden. Maybe the only choices are to bite the bullet and look into the Mesa environment (acknowledging that political choices take priority over technical ones), or to take the LRG/Alto-smalltalk route, implement some new things on Dorado, and see if they will entice others. People always seem more willing to extend things they can see than to make plans for major new directions.

start
08219 00024 US
Date: 10 MAY 1978 1737-PDT
From: DEUTSCH
Subject: Horning's reply
To: mcjones

Date: 10 May 1978 12:35 pm (Wednesday)
From: Horning
Subject: Re: Why there should be no Dorado Smalltalk
To: Deutsch
cc: CSL↑

Peter,

This is in response to your memo of May 8; I hope that you will get several others. Before addressing the substance of the memo, I would like to thank you for doing a difficult thing: publicly challenging an accepted set of premises and an established point of view (especially hard to do when they are your own). I found the fact of the memo a valuable stimulus to think about the area, and the contents themselves provocative.

After other people have had a chance to think about this issue, and respond as appropriate, I hope that you will schedule a Dealer to discuss it. I would then like to start a regular weekly series of meetings to continue the discussion, expose the issues, and explore alternative possibilities.

Let me comment on the various arguments raised in the memo before addressing the conclusions:

SMALLTALK IS CONCEPTUALLY MESSY COMPARED TO LISP:

I don't know. Lisp has a conceptually clean foundation that has by now been extended beyond all recognition. Smalltalk has a shorter history, hence may be somewhat less prone to the Topsy effect.

SMALLTALK WILL HAVE TO DUPLICATE ALL THE INTERLISP USER AIDS:

Is "duplicate" the right word? My impression (from the outside of both) is that although Smalltalk provides a DIFFERENT set of user aids, many of them approach Interlisp in power and convenience.

SMALLTALK IS INHERENTLY TOO INEFFICIENT COMPARED TO MESA:

I worry a lot less about the efficiency of experimental programs than about the difficulty of moving from an experimental to a production system. It seems clear that no small variant of Smalltalk is likely to be the language of choice for production systems in the foreseeable future. My concern isn't that modules would have to be recoded from Smalltalk into Mesa, but that from what I know of the two languages, and typical systems built in them, it seems unlikely that even the STRUCTURE could be usefully carried over--and useful structures for systems are likely to be one of the things that we try to discover through experiments.

SMALLTALK WILL NOT BE THE LANGUAGE OF CHOICE FOR BUILDING PACKAGES:

This seems likely. One wants to base experiments on efficient packages. This doesn't necessarily imply that the EPE language would have to be Mesa, but it should share a number of its underlying assumptions, so that the incorporation of packages can be reasonably easy and reliable.

SMALLTALK IS A "MOVING TARGET":

Yes, it would be hard to keep two moving targets in close alignment. This motion is both a strength and a weakness of Smalltalk, depending on the purpose for which it is evaluated. I hadn't fully realized that Dorado Smalltalk was intended as other than microcode support for the LRG system.

IF WE WANT A DORADO SMALLTALK, WE DON'T NEED TO DESIGN A NEW ONE:

Somehow, this proposal appeals to me, at least partly because it is close to my former model of what was being done.

WHAT DOES ALL THIS IMPLY?

Do I still think we need an EPE? Yes, definitely, with increasing urgency.

Does Smalltalk solve important problems for EPE? The arguments listed above reinforce a vague sense of unease I've had about the project for a long time. I don't feel comfortable with Smalltalk, for reasons that are hard to set down; maybe I just don't know enough about it. However, it is not clear to me that the strengths of Smalltalk are in the areas where EPE should be focussing.

Will people use a Dorado Smalltalk? Some, indubitably. However, you are probably right about people writing new experimental systems, even though I think that the real reason will be lack of transportability, rather than direct concern for efficiency.

Will anyone work on a Smalltalk EPE? Not as many as might be Tom Sawyered into a Mesa EPE.

More work on Dorado Lisp? Not to make me happy. I think that Lisp is at least as susceptible to the criticisms raised in Terry Winograd's "Why programming languages are obsolete" as are Smalltalk and Mesa. I see little hope of happily marrying Lisp and Mesa programs, for the same general sorts of reasons that would make it difficult to convert experimental Smalltalk programs into production Mesa programs. The underlying "linguistic philosophies" are just too different.

A "Lisp veneer" on Mesa? I'm not sure that I know what that means. I would oppose at least some of the possible meanings of such a proposal.

Alto Smalltalk on Dorado? Why not, if it's easy?

WHERE SHOULD WE GO FROM HERE?

I don't think that the idea of EPE should be lost, or allowed to founder. We need one; most of our programming withing CSL is experimental (in the EPE sense) and it is important to build tools to increase our productivity in this activity. Perhaps, however, it is time to collectively re-examine our tools, to see where the major inefficiencies are.

Some of the very-long-term issues that EPE could take a small step towards addressing are discussed very well in Terry's paper.

Jim Mitchell has repeatedly made the point that the major limitations of Mesa as a vehicle for experimental programming are not in its facilities for writing single modules ("programming in the small") but for assembling large systems out of packages of modules ("programming in the large"). This suggests that rather than attempting to use some entirely different language in EPE, and then converting successful programs to Mesa, it might be fruitful to explore the possibility of a Mesa-based EPE. Two approaches suggest themselves: Mesa as the "low-level" language to which EPE programs are translated, and EPE as a glorified binder/ configurer/recompilation system. I'm not sure what the trade-offs are here.

I returned from the latest meeting of IFIP Working Group 2.3 (Programming Methodology) more than ever convinced that our major unsolved problem is that of SPECIFICATIONS of systems and components. In fact, I think that my "ideal" EPE would have a library of three types of units: program modules in a fairly conventional language, except that each module would have "formal parameters" naming modules on which it relied, 2) specification modules, and 3) "linkage" ("module interconnection language") modules, each of which asserts that a particular program module satisfies a particular specification module, provided that its formal parameters satisfy other named specification modules. Thus one could build systems of potentially verifiable stock components, with some assurance that one knows what they will do when assembled.

One of the things that was attempted in Euclid "zones" was to allow a separation of concerns in storage management: The programmer could originally write programs without any concern for how the storage for collections was managed, and then later, if necessary, go in at a lower level of abstraction (but still within Euclid) and write his own storage management package(s). Prior to successful implementation, I doubt that anyone is fully convinced that the Euclid "solution" is completely satisfactory, but I think that it suggests a direction that EPE should explore: Top-down programming with powerful abstractions that can be given specialized implementations as and when greater efficiency is required. The EPE language should also adopt the Alphard principle that few (if any) abstractions should be preempted by the language, n the sense that the programmer is not allowed to supply specialized implementations. Guttag and Musser's DTVS (Data Type Verification System) allows one to program using only the SPECIFICATION of some data types--the system will supply a "direct implementation" that may be very inefficient, but is certain to be in agreement with the specification--and to supply efficient representations for others.

[I apologize for the length of this missive. I didn't have time to condense it.]

Jim H.

XEROX

PALO ALTO RESEARCH CENTER
Computer Science Laboratory
May 17, 1978

To Peter Deutsch
Copies to CSL, Smalltalk Interest

From Dan Swinehart

Subject Why There *Must* be an Experimental Programming Environment (EPE)

Filed as [Maxc]<Swinehart>dEpeYes.Press

For most of my professional life I've been looking for a programming environment, such as you have outlined, that could be used for building real, full-sized systems, and with acceptable performance. I even spent a few years in a previous life exploring what such a beast might look like. I agree with Jim Horning that we should discuss at length the concerns that have been raised by your memo and those that followed. But I'm writing this to emphasize how important it is that the process not take too long, and that the result be a renewed effort to get the thing built. The prior work of people named Deutsch, Kay, Teitelman, and Mitchell formed the basis for my own thesis work on interactive programming systems. We clearly have some good ideas, and now it's time to build them into something that can handle our own full-scale efforts.

For some time I've been espousing the mildly heretical view that the language we use is not nearly so important as the environment that is provided with it. Stated rather strongly, I believe a BCPL-based system surrounded by the kinds of tools proposed in the EPE documents would have stood us in better stead over the past few years than a system, using present development tools, based on any language you care to choose. Mind you, I'm not suggesting the Alto could have supported such a system. I'm also not suggesting that what we want is a bcEPEl, so to speak. We should of course give careful consideration to language choice, for formal, pragmatic, structural, and explicative reasons. I chose the example to further stress how essential it is that we also get the environment that goes with it. *Please* let us not lose this opportunity in another round of language design.

Smalltalk

To really embrace the object-oriented style of Smalltalk requires something akin to a Fourier transform in one's approach to programming; I don't even think Simula prepares one for this dramatic change in view. It is, therefore, not surprising that people find this style uncomfortable at first. Without attempting to justify the whole Smalltalk language (e.g., I'm not considering the much-diminished enforcement of data types), I have been very pleased with the clarity and compactness of expression I've been able to achieve using the object/message approach (examples available on request). In fact, I have noticed improvements when I have grafted this style onto programs written in other languages.

I have found the ability to provide multiple implementations of precisely defined interfaces particularly useful; in many cases this method served almost as a metaphor for the real-world artifacts I was simulating (and improving). And the superclass, etc., extensions to Smalltalk, recently implemented and currently proposed, are the most powerful notions for improving system structure that I've seen in a while.

Turning to some of the specific objections in the Dsmallno memo, I think the most valid one

had to do with the "moving target". Although I don't believe direct conversion of the Alto system is the right approach, because too many artifacts would remain (e.g., a virtual memory that would appear quite limited and clumsy on Dorado), we might do well to limit machine-independent extensions to the language and system that diverge from the LRG variety, at least at first. I can't imagine that the good ideas for change won't be needed in all implementations as they move into more powerful machines, so it might well be worth delaying some of them until they can be applied everywhere.

Another set of issues had to do with the Mesa connection. I don't see great difficulty converting Smalltalk packages to Mesa, especially if the object-oriented style is to be maintained (I realize there's some risk of committing Humbris again). If you believe that, the major remaining problem is access to Mesa packages from one's Smalltalk programs. It would be worth working pretty hard to make this more convenient than has been proposed. For instance, we might explore automatic or semi-automatic ways to generate a direct Smalltalk object interface to each Mesa package, with dynamic checking to bless the more flagrant of the loopholes. It looks like most of the proposed Mesa packages would do a fine job of managing the basic system resources -- disk files, Ethernet, main memory, keyboard, ... The major exception is likely to be the display facilities, which will probably be too tightly coupled to the "standard" user interface to be shared.

Those who are skeptical about the size of the Smalltalk user community are perhaps overlooking the effect that Notetakers are going to have -- probably within a year. If Notetaker and D0 programs continue to converge, this could be even more significant. I don't believe that this laboratory is so far down the Mesa path yet that it couldn't back up a bit.

Problems:

> What debugging tools will be available to debug, or at least diagnose, the Mesa packages that our systems use. Presumably the standard Mesa debugger(s), but that could get clumsy.
>
> If we decide to build the EPE around Smalltalk, who is going to build Butler's Mesa packages? Of course, some Mesa packages will have to be built to support the Smalltalk, but it will be harder to get their interfaces and functions right if they are built solely for Smalltalk. Since I estimate there is at least a Mesa's worth of work to build the EPE we ought to have, this is a serious problem.

Mesa

Before talking about the positive aspects, I want to offer a prediction that the only thing that would take us longer to build than a Smalltalk-based EPE would be a Mesa-based one.

Aside from that, I believe a reasonable interactive programming system could be built on a Mesa base. The premise here is that the system would have to be capabile of incorporating program changes very rapidly. Jim Mitchell offered a plausible treatment of most of the harder problem in his thesis. The hardest remaining problems involve the need to defeat (used either in the sense of "to conquer the problems of" or "to disable") the nearly absolute paternalism of the type system. I can imagine a compromise allowing certain liberties to be taken in the interactive mode, whose effects would later be checked more stringently by some offline process. The underlying assumption here is that incremental maintenance of the strict typing would be too expensive.

The SDD Tools group has proposed facilities that would go a long way towards providing a reasonable environment. The last version of that I saw was over a year ago, and I don't know how much of it actually got built, but it might be worth looking into. And what about Diamond?

Problems:

I'm not sure a Mesa-based EPE could ever be reflected back into an Alto or D0 environment, even to the extent that Alto Smalltalk can cope with it now.

It really is a lot of work to get anything like the kind of environment proposed for Dorado Smalltalk. There are also several very hard research problems between here and there that do not exist to the same extent for Smalltalk.

Other Languages

I refuse to consider any, for reasons already stated.

Conclusion

The importance of this effort warrants a fair amount of patience while waiting for the bandwagon to roll. I don't think lack of interest is totally responsible for the low level of participation to date. Many of us must shed at least one hair shirt before putting on another (whatever the material.) Personally, although I intend to work predominantly on voice communications, I will be looking for a way to participate in the EPE development once my printing responsibilities are under control.

During the time it takes to build a suitable critical mass, I think it's important that the project remain pointed in directions that you are comfortable with; otherwise, I don't think it has much of a chance.

Congratulations on finding a very effective method for provoking debate on an important issue!

Inter-Office Memorandum

To	CSL, SSL, Smalltalk interest	Date	May 22, 1978
From	P. Deutsch	Location	Palo Alto
Subject	Some framework for the EPE discussion	Organization	CSL

XEROX

Filed on: <Deutsch>epe-disc.memo on Maxc1

The recent exchanges of messages on the general subject of "Whither EPE?" clearly indicate that more inclusive discussion of the subject is needed. In this memo I would like to set forth a framework which I think would be useful for conducting one such discussion. The written contributions of Terry Winograd, Dan Swinehart, Jim Horning, and Larry Tesler, and conversations with Dan Bobrow, Jim Horning, and other contributors too numerous to remember, have been tremendously helpful in clarifying the issues involved.

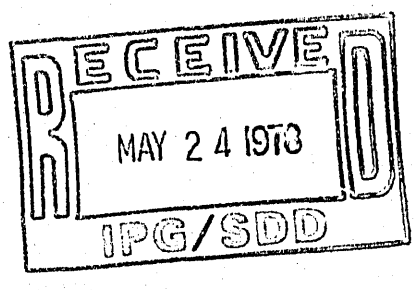
The original idea of EPE was to provide a programming environment supremely suited to the development of "experimental" programs. I now feel that there was a lot of fuzziness in what "experimental" meant. It seems to me there are four fairly distinct sets of clients whose major activities over the next few years include constructing large software systems as embodiments of research ideas. These clients are:

> The current Lisp-based community (language understanding, knowledge-based assistance, experimental data analysis, program verification, and some design automation activities).
>
> The builders of OIS function-demonstration prototypes like Woodstock, Bravo, Laurel, Draw, Markup, DDS, OfficeTalk, ... This may include some part of SDD and ASD.
>
> Those interested in radical improvements in the programming process as such (methodology, tools, language design, ...).
>
> LRG (interests in many areas, but small personal computers in particular).

I also see a variety of language and system choices available for us to put our effort into. The fundamental breakdown into Mesa, Lisp, Smalltalk, and "other" is complicated by the existence of several viable, competing Lisp dialects and by numerous plans for extensions to both Lisp and Smalltalk.

In the remainder of this memo -- which, after all, is meant to be a framework for discussion, not the conclusions of a discussion which has not yet taken place! -- I will try to set forth what I believe to be the most important characteristics of the client groups and of the language/system alternatives. I think it is necessary for us all to agree on these characteristics, at least in broad outline, before a discussion is even possible.

CLIENT NEEDS



Lisp community

The language and knowledge-based communities are building systems an order of magnitude larger than anyone else. They have a particular need for tools to help keep track of large systems being developed by many people simultaneously. Their ambitions require that they develop particular sublanguages within an overall language and system framework, and that these variant notations and structures mesh smoothly with the facilities provided in the basic system for carrying out filing, editing, printing, and analysis functions. They also have a strong requirement for transfer of a large amount of existing code, written in Interlisp and in KRL. They are concerned more with taking advantage of the horsepower of the Dorado than about wide dissemination onto less powerful machines of any systems they build.

OIS prototypes

The use of Mesa as the OIS product language requires (for "political" reasons as well as technical) that a considerable amount of the package-building activity for OIS function-demonstrations must take place in Mesa. The desire to obtain useful experience with such prototypes demands that they perform well enough that people will use them willingly, and that they run on a machine that many people will have available to them (D0 or Alto).

EPE per se

This group has been somewhat dormant over the last couple of years. Its needs include flexibility and "transparency" (minimization of hidden or unalterable mechanism) of the language system, sufficiently good performance in the things it builds that people will use them (as above), and the ability to develop many pieces of an overall system separately. Dorado is the only available machine that is likely to support a system with enough generality and acceptable performance for this group.

LRG

LRG will be working actively to develop a Smalltalk derivative for NoteTaker and D0 regardless of the outcome of the EPE discussion. However, they have a strong desire for Smalltalk compatibility or translatability in whatever we may do, a D0 implementation, and room for integration of whatever qualitatively new features they may come up with (e.g. monitors).

LANGUAGE/SYSTEM CHARACTERISTICS

Mesa

Mesa is now available on the Alto; Mesa, and a Mesa-based operating system, will be available for D0 and Dorado in the next few months. The D0 and Dorado systems will not be precisely compatible with the Alto, since the Alto does not support a virtual address space, but they are supposed to be absolutely compatible with each other.

Lisp-based

Interlisp

An Interlisp system exists on the Alto, and is being transcribed for Dorado. It has a Bcpl base and its performance may charitably be described as poor. It is about 99.2% compatible with Tenex Interlisp.

XL

Danny Bobrow and I (primarily) did a considerable amount of design for an Interlisp extension which we called XL. It has something somewhat resembling the Mesa notion of interfaces as objects, and a more complex name-to-data binding structure.

Scheme

A group at MIT has developed a Lisp dialect they call Scheme. It cleans up most of the semantic messiness present in most other dialects in a very nice way, and at the same time provides a retention mechanism that can be used to provide Smalltalk-style objects in a simple although not terribly efficient manner.

NIL

Many of the same people who did Scheme are just starting to work on a New Implementation of Lisp, for the Dec VAX and for a high-performance machine called the S1 being developed jointly by Stanford and Livermore. Not much is known about this yet.

Smalltalk-based

Current Smalltalk

The Alto implementation of Smalltalk is built on a large assembly-language base, and has severe limitations in its address space as well as some structural problems that limit its responsiveness. We are still transcribing it for Dorado, since the effort is small and the performance will be quite a bit better (although still not really acceptable for any serious clients).

Conservative extensions

A number of ideas are floating around LRG and CSL for modest, well-understood extensions to Smalltalk. The most significant ones are for extending the notion of superclass so that it becomes somewhat more like the Mesa idea of an interface, and for extending the notion of instance to multiple levels rather than a single one. Some or all of these ideas will undoubtedly find their way into the NoteTaker language.

Radical revisions

In addition to the well-understood extensions, some radical ideas are also in the air. These include restructuring of the whole language around the notion of information retrieval (or pattern matching, or dictionary lookup, depending on your point of view), and addition of a type declaration and checking system.

Other

Languages here now

Bcpl is still the dominant language at PARC, but its limitations (particularly in address space) render it unsuitable for major new projects. The AIP group has done some work in L*, and there are ideas in L* that are worth revisiting in the EPE context (such as the notion of multiple interpreters).

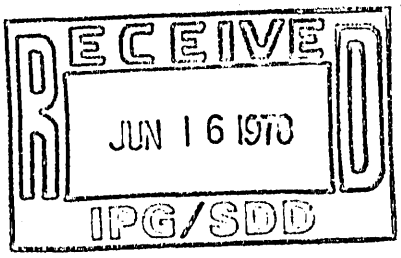
New imports

Many languages not currently in use at PARC deserve examination for the ideas they embody, if not for their immediate utility. Alphard and Euclid, among the Algol descendants, fall into this category. Discussion should reveal others.

Inter-Office Memorandum

To	CSL, Smalltalk interest, Mesa interest (Palo Alto only)	Date	June 14, 1978
From	P. Deutsch	Location	Palo Alto
Subject	Requirements for an EPE base system	Organization	CSL

XEROX



Filed on: < Deutsch > epe-reqts.memo on Maxc1

In order to work towards an EPE with significant new capabilities, we must recognize that we will have to start with a "base" system that lacks those capabilities, but has properties which will allow us to build those capabilities within it. One reason for disagreement about how to proceed with EPE is lack of clarity about the requirements that the base system must meet, and the costs involved in meeting those requirements starting from where we are now. The purpose of this memo is to set forth those requirements and to get people started thinking about the costs of meeting them.

I believe that it is important for us to remember our history and learn from it. Those readers of this document who agree with me may find it instructive to reread the memo "D1: Lisp, Mesa, and Smalltalk" (May 11, 1977; archive number 77CSL-036; file < Deutsch > D1-LMS.MEMO on Maxc1).

I have not set priorities among the requirements because failing to meet *any* of them will have major adverse impact. "Lisp" means Alto Lisp, and its reimplementation for the Dorado under the Alto emulator; "Mesa" means Alto Mesa 4.0, and its reimplementation for the Dorado and D0; "Smalltalk" means Alto Smalltalk, and its reimplementation for the Dorado under the Alto emulator. Square brackets indicate costs for repairing, enhancing, or providing an immediately previously mentioned facility: if filled with a man-month cost or a target date, they represent an estimate supplied by the system implementors; if empty, they represent an unanswered question.

Language and Virtual Machine

Requirements

Virtual machine

(1a) Large address space: at least as large as the hardware address space (initially 2↑22 words, eventually 2↑24).

Language

(1b1) Adequate collection of built-in data types; ability to define new ones; class mechanism; type checking.

(1b2) Automatic storage management (reference counting or garbage collection; stack allocation); guarantee of no dangling references.

(1b3) Multiple name scopes.

Packages

(1c) Reasonable access to terminal (display and user devices), Ethernet, disk files.

Performance

(1d) On the Dorado, several times better than Maxc for the things currently being done in Lisp.

Costs

(1a) Lisp already meets this criterion, although some reimplementation is needed to take advantage of the Dorado's addressing hardware [< 1]. Mesa 4.0 contains language and instruction set provisions for a 24-bit address space, and the compiler can generate these instructions, but the runtime system and standard packages are not prepared to handle long pointers [] and there is no memory management system for the D* that would make them useful [Pilot 2.0: 8/30/78]. Smalltalk has an address space of 2↑16 objects, which represents around 2↑20 words: increasing this would require major system changes [6].

(1b1) All 3 languages have a reasonable set of built-in data types. (Mesa lacks in-line floating point [], but is more complete in other ways.) Lisp has a record facility similar to that of Mesa's without variants, but it is not implemented in the Alto/D* system []. Smalltalk's class mechanism is the most flexible semantically, but the least efficient in storage [2]. Neither Lisp [] nor Smalltalk [] has compile-time type checking.

(1b2) Lisp has two garbage collectors: one fast-acting and based on reference counts, the other slow-acting and based on copying. The latter does not exist []. Mesa has no provision for automatic storage management at all []. Smalltalk uses reference counting. Mesa provides for stack allocation; Lisp does not []; Smalltalk doesn't need it because of its reference counting strategy.

(1b3) Mesa has an extensive scoping mechanism. Lisp has one level of local naming for variables, a single global space for free variables [], and a global space for functions except for block compilation []. Smalltalk has a 4-level scope mechanism for variables [] and class-relative operation names.

(1c) Lisp has a very complex and poorly documented [] set of facilities for accessing the terminal in a manner that assumes it is located remotely; no facilities for using the Ethernet []; a reasonable (but only sequential []) interface to the Alto file system. Mesa has good packages for all the devices -- several incompatible ones for the display. Smalltalk has good access to the terminal, a rather primitive package for the Ethernet (supporting only the basic Pup protocol and EFTP) [2+ for FTP], and good access to files.

(1d) Some very rough estimates suggest that Lisp will run on the Dorado at around 2 x Maxc speed. Smalltalk should be comparable. No comparable estimate is available for Mesa, although comparison of microcode suggests that Mesa will run 3 to 5 times faster than either Lisp or Smalltalk.

SYSTEM

Requirements

Tools

(2a1) Ability to modify program and resume execution with minimal loss of state.

(2a2) Very low time overhead for program modification (< 5 seconds).

(2a3) Hooks for debugging, monitoring, performance measurement.

Reliability

(2b1) No failure modes that destroy source code.

(2b2) At least the ability to create an undestroyable checkpoint in < 30 seconds.

Costs

(2a1) Lisp and Smalltalk both provide this facility -- often, no loss of dynamic state is incurred at all. Mesa currently requires complete loss of dynamic state [].

(2a2) In Lisp, the time overhead is that required to get into the editor and out again -- a matter of typing about a dozen characters. In Mesa, editing, compilation, and binding are a batch process requiring on the order of minutes for a program of significant size []. In Smalltalk, the overhead includes finding the method to be edited with the Browser (1 to 10 seconds) and recompiling (3 to 10 seconds on the Alto).

(2a3) Lisp includes BREAK and ADVISE facilities, and a fair amount of existing system machinery for using them. Mesa has good debugging and measurement facilities built into the system, using special hooks in the microcode and knowledge about the form of compiled code. Smalltalk has a simulator and a PC sampler for measurement, and not much else [].

(2b1) Any reasonably serious system failure in Lisp [] or Smalltalk [1-2] will wipe out all source changes since the last manually-initiated checkpoint operation. This is not a problem in Mesa since editing is done with Bravo.

(2b2) Lisp provides a checkpoint facility which copies the entire virtual memory file. Smalltalk has no such facility, but it is possible to copy the file by going out to the Alto Executive ["no way" to fix]. Mesa has a checkpoint facility equivalent to that in Lisp. None of the three systems can recover from a checkpoint if any files open at the time of the checkpoint have moved on the disk [, ,].

|

SUPPORT

Requirements

Documentation

(3a1) Complete documentation for the language, at least in reference form.

(3a2) Enough documentation so that the user of a package or tool doesn't need to read the code.

Transportability/compatibility

(3b) Must be available in absolutely compatible form on both Dorado and D0, with a speed ratio of no worse than 5 to 1.

Politics

(3c1) Must not make a research organization dependent on a development organization, or vice versa.

(3c2) Someone must make a long-term commitment to maintenance.

Costs

(3a1) The Lisp "base language" is very simple and well-documented; however, the CLISP extensions are poorly defined, do not have a describable syntax, and are only moderately well documented []. The documentation for Mesa is very good. The Smalltalk language reference documentation is also very good; however, since the meaning of almost all Smalltalk operators depends on the operand types, a lot of what one would think of as the "language" is determined by the system classes, which are not nearly as well documented [].

(3a2) This is true of most, but not all, of the Lisp facilities []. In Mesa the quality of documentation also varies widely from one package to another []. The Smalltalk documentation for packages is spotty [2], and even for some parts of the language [1], but since all the source code is instantly available on-line, having to read the code is less onerous than in the other systems.

(3b) Lisp [] and Smalltalk [3] will have this property, assuming it is true of the Alto emulator, if D0 microcode is written. Mesa is supposed to have this property, running under the Pilot operating system, but Pilot does not exist for either the Dorado [] or the D0 yet [Pilot 2.0: 8/30/78].

(3c1) Lisp support is provided entirely within PARC, with some help from BBN []; Lisp has no Xerox clients outside PARC, although we have an Arpa contract to support it for outside users. PARC is currently entirely dependent on SDD for Mesa support, and there are no plans for changing this []. Smalltalk support comes from LRG; SDD is a minor client [].

(3c2) Currently, Warren Teitelman and some people at BBN have a long-term commitment to maintain Interlisp; there is no one committed to maintaining the PARC Lisp systems over a long period []. SDD is committed to maintain Mesa indefinitely. LRG is explicitly not committed to maintain Smalltalk as it currently exists for more than 6 months or a year, and no one else has volunteered [].

Inter-Office Memorandum

To	CSL, Smalltalk interest, Mesa interest (Palo Alto only)	Date	June 13, 1978
From	P. Deutsch	Location	Palo Alto
Subject	E(PE) vs. (EP)E	Organization	CSL

XEROX

Filed on: <Deutsch>epe-vs-epe.memo on Maxc1

Here are some alternatives we should keep in mind as we consider where to put our effort in the area of programming environments.

E(PE) vs. (EP)E

Building systems with current, demonstrated technology, for use by people who want to do experimental programming in a variety of areas (more on this below).

Developing new programming-system technology.

Kinds of users

Knowledge-intensive: KBA, natural language, other KRL applications.

OIS functional demonstrations: DDS, Star mockup, current state of Juniper.

OIS usable prototypes: Bravo, Laurel.

Data-intensive: design automation, IDL, conventional DP.

Step size

Build new facilities within existing system environments.

Build new environments using current languages.

Build new languages and systems.

Hardware environment

Build for Dorado in a manner that fully exploits its power, without expecting transferrability to a smaller system.

Build for Dorado in a manner that anticipates transfer to a smaller system such as D0.

Build for D0 and/or NoteTaker.

Some current technology

Mesa type and scope rules

Lisp/Smalltalk automatic storage management

Mesa and Lisp tools for consistent compilation

Smalltalk browser

Masterscope

Mesa Librarian

Lisp inter-person coordinator (NAMEFNS)

Lisp multi-language capability

Some interesting PE research questions

Cooperative development of programs.

Programs as data bases; propagation and management of changes.

Multi-language systems; multiple interpreters; less distinction between program and data.

Configuration description, version control.

Modest but useful formal aids (verification).

New, powerful language concepts.

Understanding the performance behavior of programs.

How to factor descriptions of programs better.

Before we even consider alternative paths, we must ask what goal we are trying to reach. I assume we are ultimately concerned with bringing computer power in useful form to as many people and applications as possible. Thus we are always considering functional capability (how can a computer perform some function at all), usability (how can we present this capability in a way that meshes well with people's thinking), and cost (both to the end user and to us) together. Because we are presumably working in areas largely unexplored, the cost to us of doing experiments is substantial and must figure in our plans. Furthermore, as a research organization I think we have a particular need to choose our projects to maximize what we *learn* about the technology, as opposed to a development or market probe organization. Of course there are different ways to learn: a system that works at all teaches us something about functional capability, but a system must work well enough to attract users to give us much data about usability.

I believe we have three basic alternatives before us in the area of programming environments, and the choice between them depends on our value system, not on our technical judgement. We can concentrate on making the Mesa environment more pleasant and productive for us (and, of course, SDD); we can do some work on programming environment per se in the context of Lisp; or we can look for a new language and system that

brings together the best ideas that have come out of Lisp, Mesa, Smalltalk, and the other languages and systems we have encountered over the years.

I don't think we have other credible alternatives. The Mesa language and system are so complicated already, compared to Lisp or Smalltalk, that just coping with this complexity would be a large hindrance to anyone interested in programming systems as such (e.g. Larry Masinter): for this reason I don't think PE work, other than of a very incremental nature, can be done in the Mesa framework. There is no prospect that Lisp or Smalltalk can be made efficient enough, aside from redesign that would essentially create a new language, to compete seriously with Mesa for usable functional prototype work. The extent of existing tools and packages in both Lisp and Mesa so dwarfs the corresponding Smalltalk facilities that there is little to choose between Smalltalk and a new language system in this regard.

I am very tired of trying to discuss alternatives in a vacuum. The only concrete work on EPE going on right now is the SDD work on Mesa tools and CSL's work in the Lisp/KRL environment. I still think neither Mesa nor Lisp is the right model for the programming environment of the future, and I don't think Smalltalk is right either: I do think there are a set of ideas that could be synthesized into a language into which Lisp, Smalltalk, and a very large subset of Mesa could be translated mechanically, and in which the choices between the three different styles or world-views would appear as efficiency tradeoffs in the design of the instruction set and storage management system. I don't think much, if any, existing system code could be re-used, although a lot of user code probably could be, and I don't think that's important because I think a well-designed system that doesn't try to be all things to all people can deliver a lot of capability without a tremendous amount of machinery. I think we should tailor this new system for the Dorado and not worry about mapping it anything smaller, because by the time we've learned enough from it for it to be suitable for a Mesa-like transfer (or redesign) the technology will have caught up. I think we should be looking at applications in which complexity of structure, rather than volume of data, is the primary bottleneck: thus, I envision a system which would eventually be good for KBA and for OIS prototypes but maybe not so good for the data-crunching parts of design automation. The system would have to be good enough pretty soon to attract a few users, but its primary purpose would be to develop programming technology, and not deliver a service as such for a long time.

That's what I think, and that's what I intend to do.

Inter-Office Memorandum

To	Distribution	Date	July 13, 1978
From	Ed Satterthwaite	Location	Palo Alto
Subject	Prospects for Mesa in SD	Organization	SDD/SD

XEROX

Filed on: [Iris] < Satterthwaite > MesaProspects.Bravo

Here are my guesses about what SDD will and will not do to develop the Mesa system over the next few years. They are based on discussions I had with David Liddle before joining SDD and my readings of current trends, but I emphasize that they are personal opinions and not promises by anyone.

General Comments

SDD has embarked on an ambitious product program. Further development of Mesa won't offer very much immediately visible leverage in getting the product out, so it's reasonable to expect a constant or diminishing commitment of resources to Mesa per se. In particular:

In the short term, any incompatible changes to the virtual machine or language will be strongly resisted unless the benefits of such changes are very substantial and very obvious. In the longer term, SDD should be interested in helping to design and implement significant extensions in these areas when they address acknowledged problems of interest to SDD (development of the process stuff provides a good model of how this might happen).

A lot will be done to improve the performance of all components of the Mesa system. I believe that the compiler, for example, will respond well to tuning and cleanup, especially on the Dstar.

For obvious reasons, there is interest in more extensive optimization (both speed and space), particularly of the Tools B2 flavor, and in static analysis that is likely to reveal lurking bugs (e.g., range analysis, uninitialized variable detection). I am personally interested in doing these sorts of things.

A lot of effort will go into tools. My feeling in the past has been that SDD's tools are likely to be somewhat heavy-handed in providing administrative control; I have come to appreciate the controls for large, multi-person projects but am not convinced I would like them for a one or two person experiment.

The products will clearly provide many of the functions we would like in packages, but it is unrealistic to expect packages from the product program except by luck (it might be possible to extract and repackage chunks of code). Collaborative development of packages with the prototype software group (who also need an (EP)E) seems more likely and promising.

Mesa 5, 6, ...

There are tentative plans for Mesa 5 (early 1979) and sketchy plans for Mesa 6 and 7. Mesa 5 will concentrate on providing "major" language features that are well enough understood to be ready for implementation and that we hope will impact the organization of the product software in desirable ways. I think this means

inline procedures

defaulted record fields and arguments (constants only)

sequences (generalizations of strings), cleanup of variant records, and some associated machinery for dealing with variable-length objects a bit more cleanly.

some provision for decoupling source and object packaging of code (to get reasonable working sets - not very well understood yet)

possibly some support for object-oriented programming (current proposals need more analysis and discussion, but, e.g., Peter Bishop's memo contains or suggests some subproposals that appear useful and less controversial).

Comments on the Features List

I use Jim Horning's numbering.

Virtual Machine / Programming Language

Query the Pilot group also on virtual machine issues.

(A4) Object management -- garbage collection/reference counting

I think everyone is in favor of more hassle-free storage management, but I wouldn't count on much. The best I can imagine without massive (and incompatible) reorganization of existing SDD system designs is some way of overloading assignment to do reference counting.

(A8) User access to the machine's capability for packed data

Algorithms for packing (non-machine dependent) records will be further improved. Bit arrays seem to come almost for free from the instruction set and will probably appear.

(A9) Program-manipulable representation of programs
(A10) Run-time availability of all information derivable from source program (e.g. names, types, scopes)
(B3) Ability to create fully integrated local sublanguages.

Our discussions of local sublanguages revived fond memories of the Tenex MPS debugger. I would like to extend the current debugger's interpreter and extract it as a user-callable package; it doesn't look terribly hard.

(A12) Self-typing data (a la Lisp and Smalltalk), run-time type system

Sequences, cleaner variant records, and some help with allocation/initialization are planned and mostly designed. Object-oriented programming is recognized as a good thing and appears throughout the design of the product software. Any good proposals for language support of "objects" would probably be accepted and supported by SDD.

(A14) Encapsulation/protection mechanisms (scopes, classes, import/export rules)
(B7) Control over importation of names

The "redeclaration problem" (especially using the include + OPEN mechanisms) has been the source of a number of subtle bugs and causes problems in the design of, e.g., DeSoto. We plan to reexamine controlling the importation of identifiers and/or their redeclaration.

(A16) Non-hierarchical control (coroutines, backtracking)

There is a modest commitment to making ports useful.

(B5) Good facilities for processes, monitors, interrupts

Initial experience suggests some modest adjustments and extensions in these areas.

(B6) Simple, unambiguous syntax

With a prettyprinter, we hope it will be possible to clean up some of the known botches.

(A2.1) Well-integrated access to large, robust data bases
(A18) Inter-language communication
(B1) Inheritance/defaulting (Smalltalk subclassing)
(B2) Ability to extend language (e.g. operator overloading)
(B8) Closures
(B9) Full-scale inter-language communication
(C1) Clean data and control trapping mechanisms
(C2) "Good" exceptional condition handling

Don't expect much in these areas, either because the feature has limited perceived value (e.g., A18) or appears to have too much research content. A complete, plausible design for something like Mitchell-Wegbreit schemes seems highly desirable and could probably count on SDD for implementation help.

Tools

Query the Tools group also on these issues.

(A1) Fast turnaround for minor program changes (<5 sec)

This is considered absolutely essential and should not be hard to achieve on Dstar. Note that DeSoto and the ability to replace modules in a system that has already run are planned and will help.

(A2) Compiler/interpreter available with low overhead at run time

See the comments on A9, etc. above.

(A3) Masterscope, cross-reference, annotator

There are various batch cross-reference programs; better versions will appear.

(A4) Consistent compilation
(B6) Version control

These are considered essential. DeSoto and (to a lesser extent) the Tools Librarian make a first cut at this. There is a working group trying to understand the remaining issues and make recommendations.

(A5) Source-language debugger

Current developments are directed toward cross-debugging, i.e., debugging programs on another (virtual) machine using a very narrow interface. I hope that time will also be found to clean up the command language and improve the performance.

(A6) Dynamic measurement facilities

These are considered essential. Many of the primitives exist but lack packages to make them useful.

(B2) More optimizing compiler if user willing to bind more tightly -- with full compatibility
(C2) Static analyzers: verifier, performance predictor

See the introductory remarks. Inline procedures and packaging are first stabs at B2.

(B3) Librarian, program-oriented filing system (incl. Browser)

The Tools Librarian provides the underpinnings for this.

(B4) Prettyprinter

This, is considered essential and will appear.

Packages

Much of the desired capability has appeared or will appear soon. I am not prepared to guess what will be usable in a PE in terms of observance of conventions, packaging and documentation.

Inter-Office Memorandum

To	Distribution	Date	July 21, 1978
From	PE working group (names on next page)	Location	Palo Alto
Subject	First report from Programming Environment Working Group	Organization	CSL

XEROX

Filed on: Ivy <Deutsch>per>
report>* - the report proper
future>* - the memos which were condensed into section 5b
minutes>* - the minutes of the working group's meetings

Distribution:
CSL
SSL/LRG
SDD Palo Alto: Irby, Liddle, Lynch, Metcalfe, Wallace, Weaver, White, Wick
ASD Palo Alto: Brotz, Elkind, Spitzen, Wegbreit

Non-CSL recipients may distribute further copies as needed

Table of Contents

1. Charter and history
2. Introduction, rationale, and conclusions
3. Catalog and discussion of PE capabilities
    a. Virtual machine and programming language
    b. Tools
    c. Packages
    d. Other
4. Priority ranking and interrelation of capabilities
5. Relation to existing environments (Lisp, Mesa, Smalltalk)
    a. Analysis of difficulty of high priority items in each environment
    b. Projections for non-EPE development

Charter and History

The purpose of this report is to collect and set down CSL's experience and intentions in the area of programming environments. Some eventual version of the report will reflect the contributions of all interested members of CSL, and other appropriate input from the rest of the Xerox Palo Alto community. This version, however, has been produced by a working group consisting of the following people:

Peter Deutsch
Jim Horning
Butler Lampson
Jim Morris
Ed Satterthwaite (SD)
Warren Teitelman

with the occasional participation of Al Perlis (consultant).

The original charter of the working group was to fill in an outline produced in consultation with Bob Taylor, reproduced below with Bob's high-priority items in boldface. We quickly decided that the right way to proceed was to address what we felt was the real question requiring resolution, namely to produce a catalog of programming environment capabilities which included justified value, cost, and priority evaluations. Even though we did not attempt to fill in the original outline as such, our results can be fitted into it as indicated by the italicized text interspersed with the outline.

The working group was given a one-month deadline and held 8 two-hour meetings. Needless to say, there were many areas we had to treat superficially because of the time constraint, and some areas in which we realized we simply could not reach agreement. We also realize that we have undoubtedly overlooked some significant issues and viewpoints. However, we expected much more in the way of intractable disagreement than we actually experienced. This suggests that we were successful at avoiding "religious" debates, and concentrating on the technical issues.

I. Introduction, Charter, History

A. What have we learned from Interlisp, Smalltalk, and Mesa that is important?

There is a collection of capabilities that we know from experience are of significant value in a programming environment. Each of these systems has contributed to this experience. No one of them possesses all these capabilities. In addition, the degree to which the capabilities are integrated into the system can have a profound effect on their utility.

B. What can we learn from Interlisp, Smalltalk, and Mesa that is important?

We did not really address this question. It might be possible to learn more about the value of certain capabilities by actual measurements of the use of these systems, but we think that (a) no one will feel motivated to do such measurements, and (b) the synergy between features makes it very difficult to measure their individual values.

II. Definition of Programming Environments

III. Survey of Local State-of-the-art

The local state-of-the-art is captured implicitly in section 5a.

A. Interlisp

B. Smalltalk

C. Mesa

D. Summary and catalog of proven features (what we already know)

This is the main body of the report, sections 3 and 4.

IV. Research Issues

A. Evaluating programming environments

As mentioned above, we believe that anecdotal, personal experience is our only realistic way to evaluate programming environments at present.

B. Preservation and transmission of lessons learned

Aside from the report itself, the main vehicle for transmitting the lessons learned will be the improvement of whatever environment(s) we choose for further development.

C. New ideas worth investigation in the Interlisp, Smalltalk, and/or Mesa environments

This is implicit in sections 3, 4, and 5a.

D. Selecting environmental features

E. Integrating the environment

V. Tool issues

A. CSL requirements for programming environment(s)

This is captured by sections 3 and 4.

B. Deficiencies in present CSL programming environments

This is implicit in the high-ranking items which do not exist in our present environments.

C. Tactics for meeting CSL requirements

We deliberately did not address this point, feeling it was premature before the entire lab had a chance to comment on the report. Section 2 has some comments on this.

D. Research issues in tool construction

VI. Resource Issues

In general, we did not address these issues, again because we felt it would be premature. Sections 2 and 5b touch on them indirectly.

A. What would various approaches cost?

We discuss the difficulties of implementing individual features in detail in section 5a, but only touch briefly on the global issues.

B. What should be CSL priorities?

C. A long-range plan for programming environments

D. Reasonable first steps in the next two years

Introduction

How should we compare programming environments?

Before considering particular features that we feel contribute to "good" programming environments, it is important to consider how we can tell that one programming environment is "better" than another.

> Any evaluation must be in some context. There is no reason to believe that some one programming environment could be optimal for all kinds of programming in all places at all times. In our discussions, we have focussed our attention on the foreseeable needs within CSL in the next few years, with particular attention to "experimental programming." We have taken experimental programming to mean the production of moderate-sized systems that are usable by moderate numbers of people in order to test ideas about such systems; Bravo and Laurel were seen as typical of such experiments, but we believe that it will be important to conduct future experiments more quickly and at lower cost.

It is difficult to quantitatively compare programming environments, even in fixed contexts. A large number of qualitative comparisons are possible, but the more convincing ones all seem to fall into two categories, both based on the premise that the purpose of a programming environment is, above all, to facilitate programming:

> A good programming environment will reduce the *cost* of solving a problem by software. The cost will include the time of programmers and others in design, coding, testing, debugging, system integration, documentation, etc., as well as of any mechanical support (computer time, etc.). Since our human costs continue to be dominant, one of the most convincing arguments in favor of a particular programming environment feature is that it speeds up some time-consuming task or reduces the need for that task (e.g., it might either speed up debugging or reduce the amount of debugging needed). Bottleneck removal is the implicit argument in much that follows.
>
> A good programming environment will also improve the *quality* of solutions to problems. Measures of quality include the time and space efficiency of the programs, as well as their usability, reliability, maintainability, and generality. Arguments relevant to this category tend to be of the form "this feature reduces the severity of a known source of problems," or "this feature makes it easier to improve an important aspect of programs." Thus a feature might be good because it reduces the frequency of crashes in programs or because it makes it convenient to optimize programs' performance.
>
> (These two categories could be reduced to one by noting that there is a tradeoff between cost and quality. Thus by fixing cost, we could compare only quality; by fixing quality, we could compare only cost. This seems to complicate, rather than simplify, a qualitative evaluation of features, so we will not seek to unify these two kinds of argument.)

In the discussion that follows, we have attempted to relate our catalog of "important features" to these more general concepts of what makes a "good environment" whenever the connection is not obvious. In some cases, we have not been entirely successful, because our experience "tells" us that something is essential, but we haven't been able to analyze that experience to find out *why*. In all cases, our arguments are more intuitive than logically rigorous. Strengthening these arguments would be an interesting research topic.

We have been largely guided by experience with three current programming environments available to the PARC community: Interlisp, Mesa (with Bravo), and Smalltalk. Both what we know about the strengths and what we know about the limitations of these environments have been taken into consideration. It is of course dangerous to generalize too boldly from the intuitions and preferences of users: it is virtually impossible to be certain that the useful features have been clearly distinguished from those that are merely addictive.

Conclusions

We are somewhat surprised at how little disagreement remains in our rather mixed group. Our priority ranking of PE features (see section 4) is numerical evidence of this, and the minutes of our discussions (on [Ivy]<Deutsch>per>minutes>*) provide further, qualitative support for the same point. There are, of course, differences about the relative importance of many features, but hardly any about the five-level priority assigned to each feature, or about how hard things are to do. We therefore urge that discussion should focus on the value of an EPE to CSL, and how it is to be achieved, rather than on detailed argument about individual features.

As we said above, the purpose of a PE is to improve the productivity of programmers or the quality of programs. We don't know how to quantify quality, but we did think about how much more productivity we might expect for sizable projects (e.g. Laurel, Bravo or the travel planner), as compared to the current state of affairs in either Lisp or Mesa. We guess that a factor of four is possible, about half from relaxing current space and time constraints by moving to the Dorado, and half from a PE which has our A, B, and C priority features, and a respectable sprinkling of the others.

How will all this productivity be applied? We anticipate three major effects.

> First, many more interesting things will be within the scope of a single person's efforts. Hence, the number of Laurel-sized projects can be expected to increase dramatically; not only are they much less work to do, but it is much easier to organize a one-person that a four-person project.
>
> Second, much more elaborate things will be feasible, and hence will be attempted.
>
> Third, the evolution of good packages which can be easily used without disastrous time, space or naming conflicts will cause a qualitative change in the nature of system-building: the current Interlisp system gives us our few hints of what this change will be like.

What next? We think that CSL can support only one major PE, and that it can be evolved from either Lisp or Mesa; Smalltalk is farther from our goals, though certainly not out of sight. If a new PE is to be developed from Lisp or Mesa, then other work on the existing systems must be kept to a minimum during this development (unless we want half the lab's effort to be devoted to PEs), and they must be phased out as the new environment becomes viable. There doesn't seem to be any reason to attempt a multi-language PE, since an environment with all our ABC features would support any existing style of programming, while a two-language system would require more work, reduce synergy, and create artificial barriers. Our problem, then, is to decide whether we want to pay the price for a new programming environment:

> agreement on what the starting point should be;
>
> manpower to build the ABC features which are missing;
>
> willingness of users to suffer substantial inconvenience in changing over.

We are clearly talking about a project on the scale of the Dorado, and we should give it just as much careful thought as we did the Dorado project. Lab-wide discussion, and written feedback, seem like appropriate ways to carry out this process.

Catalog of Programming Environment Capabilities

We have divided the capabilities of a programming environment into four categories. "Virtual machine / programming language" refers to those capabilities that are primitive concepts in the programming language or in the virtual machine on which the programming language runs. "Tools" refers to capabilities for operating *on* programs. "Packages" refers to readily available programs that implement particular clearly defined higher-level concepts. "Other" includes documentation and non-technical considerations.

It is important to note that the division between tools and packages is a somewhat arbitrary one: in a good environment, there is little distinction between the two in that all the capability provided by the tools is available to the programmer in the form of packages, and all the capability of the language (including the packages, of course) is available to the terminal user to extend the functions of the tools.

VIRTUAL MACHINE / PROGRAMMING LANGUAGE

(L1) Large virtual address space ($\geq$ 24 bits)

(L2) Direct addressing for files

  (L2a) Segmenting

  (L2b) An enormous virtual address space ($\geq$ 48 bits)

(L3) Well-integrated access to large, robust data bases

(L4) Memory management -- object/page swapping

(L5) Object management -- garbage collection/reference counting

(L6) Some support for interrupts

(L7) Adequate exceptional condition handling

(L8) User access to the machine's capability for packed data

(L9) Program-manipulable representation of programs

(L10) Run-time availability of all information derivable from source program (e.g. names, types, scopes)

(L11) Statically checked type system

(L12) Self-typing data (a la Lisp and Smalltalk), run-time type system

(L13) Encapsulation/protection mechanisms (scopes, classes, import/export rules)

(L14) Abstraction mechanisms; explicit notion of "interface"

(L15) Non-hierarchical control (coroutines, backtracking)

(L16) Adequate runtime efficiency

(L17) Inter-language communication

(L18) Uniform screen management

(L19) Inheritance/defaulting (Smalltalk subclassing) (difficulty depends a lot on how much it has to do)

(L20) Ability to extend language (e.g. operator overloading)

(L21) Ability to create fully integrated local sublanguages

(L22) User access to the machine's capability for multi-precision arithmetic

(L23) Good facilities for processes, monitors, interrupts
(L24) Simple, unambiguous syntax (including infix notation)
(L25) Control over importation of names
(L26) User packages as "first-class citizens"
(L27) Closures
(L28) Full-scale inter-language communication
(L29) User microprogramming
(L30) Clean data and control trapping mechanisms
(L31) "Good" exceptional condition handling

TOOLS

(T1) Fast turnaround for minor program changes (< 5 sec)
(T2) Compiler/interpreter available with low overhead at run time
(T3) Cross-reference/annotation capability
(T4) Prettyprinter
(T5) Consistent compilation
(T6) Version control
(T7) Librarian, program-oriented filing system (including Browser)
(T8) Source-language debugger
(T9) Dynamic measurement facilities
(T10) Checkpoint, establishing a protected environment
(T11) History and undoing
(T12) Editor integrated with language system
(T13) More optimizing compiler if user willing to bind more tightly -- with full compatibility
(T14) Aids for incremental development (stubs, outstanding task list)
(T15) Regression testing system
(T16) Random testing aids
(T17) (high capability) Masterscope
(T18) Access to on-line documentation (Helpsys)
(T19) Static analyzers: verifier, performance predictor

PACKAGES

(P1) Text objects and images
(P2) Line objects and images
(P3) Scanned (bitmap) objects and images
(P4) Press files
(P5) More elaborate screen management
(P6) Remote file storage

(P7) Small data base manager
(P8) Message transmission system
(P9) Remote procedure call
(P10) Event logging
(P11) Background processing
(P12) Generalized cache
(P13) Document editing
(P14) Forms
(P15) Menus & other standard user interfaces
(P16) History lists
(P17) User access to full bandwidth of disk
(P18) (English) dictionary service
(P19) Teleconferencing
(P20) Audio
(P21) User access to full bandwidth of networks

OTHER

(X1) Adequate reference documentation
(X2) CSL control over the system's future
(X3) "Efficient" interface for experts
(X4) Uniformity in command interface
(X5) "Self-teaching" interface for beginners
(X6) Good introductory documentation

VIRTUAL MACHINE / PROGRAMMING LANGUAGE

(L1) Large virtual address space ($\geq$ 24 bits)

(L2) Direct addressing for files

(L2a) Segmenting

(L2b) An enormous virtual address space ($\geq$ 48 bits)

(L3) Well-integrated access to large, robust data bases

The issue here is that things should scale smoothly as programs grow to encompass more functions or larger data bases. As things stand, the *time* required tends to grow in predictable ways, but when the *space* (addressing or memory) requirements grow beyond a certain point, radical redesign of the program is usually required. A secondary issue is the ability to combine programs without running into the same kind of hard constraint on the space taken up by the code.

An address space of, say, 2↑24 items would be adequate to hold all the code actually being used even in a large system, but not adequate for all of its data (e.g. the American Heritage dictionary), whereas an address space of, say, 2↑48 items *would* be adequate for all the code and data of a very large, even multi-machine system. We tend to favor the former, more conservative definition, since we do not understand how to provide an efficient, robust implementation of the latter (more on the robustness question below). However, we feel that it is essential that a transition across the 2↑24-object boundary not require the kind of wholesale reorganization of programs that such a transition requires in current language systems.

System facilities for accessing large, external data bases are required for several reasons:

> Many questions of organization and efficient implementation can be solved once, rather than over and over again by applications.
>
> The system itself needs data base facilities for tools like the librarian.
>
> Access to externally stored data objects needs to be smooth for the debugger and other system facilities, not just the application programs.

Programs normally refer to internal objects with individual references, but to external data bases with both individual references and "mass" queries. There are three basic techniques for speeding up references to external data:

> Caches (good for individual references);
>
> Using sequentiality properties (good for searching);
>
> Inversion (good for searching).

All of these techniques should be available in package form.

Integrity

A programming environment in which the file system is viewed as an extension of the address space must be extremely robust -- much more so than any programming system we now have. Our current practice is to make the "truth" for a data base be a text file, and not to expend a lot of effort on armoring the binary version against all imaginable errors. Notable exceptions are the basic file facilities like IFS; we believe that a system that provides robustness facilities usable at higher levels would have a lot of advantages.

Long-term integrity seems to require some kind of history or redundancy information.

> To protect against "cosmic rays", it is sufficient to record this information in a form that only the implementing program understands.
>
> To provide Undo capability, the history must be in a form that makes sense to clients.

Another kind of integrity has to do with not losing information. The following kinds of ideas are important in this connection:

> A computed object (e.g. a Mesa configuration) should keep track of how it got made, in enough detail to make it again. The description of the putting-together process should be program-manipulable.
>
> Files should be self-identifying in a way that does not depend on their transfer between storage media or locations.
>
> It should not be possible to destroy all traces of a file containing input (as opposed to purely computed) information. The space requirements can be kept under control by storing descriptions of files (such as changes from other files) rather than all the bits.

Even the dangling reference problem and various garbage collection approaches can be viewed as integrity questions to some extent.

(L4) Memory management -- object/page swapping

(L5) Object management -- garbage collection/reference counting

These facilities are essential for essentially the same reasons as (L1) through (L3), namely, to free programmers from excessive concern for the size and location of their code and data, for both explicitly named and dynamically constructed objects. Pilot will take care of our memory management needs if we use page swapping. Alternative swapping strategies, such as Ooze, were not discussed: they are most applicable to environments that are very short on high-speed memory, which we believe will not be the case on the Dorado.

(L6) Some support for interrupts

The ability to interrupt program execution is essential, at least to gain control of runaway programs, to allow interaction with long computations, and to provide local multiprocessing; "good" facilities are less important. We have no strong religious feelings about the form of a process mechanism.

(L7) Adequate exceptional condition handling

An integrated mechanism for handling exceptional conditions is required for clean debugging and for the construction of robust programs; it helps clarify program structure by separating "normal" and "exceptional" algorithms. Certain mechanisms are required in this area: some way to unwind the stack, some way of catching an unwind, and program control of error processing. What should be provided beyond this is not agreed -- the controversies raging around the Mesa signal facilities are a reflection of our poor understanding of the problem.

(L8) User access to the machine's capability for packed data

The ability to pack data is essential to obtain acceptable storage efficiency for large data structures. However, in a system without static (pre-runtime) type checking, the desire to access packed structures efficiently (without checking or indirection through descriptors) conflicts with the desire to prevent corruption of the storage management system. There was a long discussion of why it is hard to add packed data to Alto Lisp, which centered around this protection question in the following guise: how carefully should the system prevent the user from smashing its underlying data structures. There wasn't much agreement on this point, but it does seem to have considerable practical importance, since a highly restrictive attitude makes it difficult to code low-level parts of the system in itself.

(L9) Program-manipulable representation of programs

(L10) Run-time availability of all information derivable from source program (e.g. names, types, scopes)

These two issues are closely related: underlying them is our desire to make it easy to extend the set of tools, to make communication between sublanguages easy, and to break down as many as possible of the artificial distinctions between programs and their compilers, on one hand, and data and their programs, on the other. A long discussion led to the conclusion that all this information is currently available in Mesa, modulo the question of how stable the compiler's internal representation of the program as a tree should be expected to be. Straightforward methods (like the ones used in Clisp) will allow compiled code to be attached to source in a user-created data structure, although it is certainly easier to do this in Lisp, where interpretation is simple. Not explored was the usefulness of an interpreter for a subset of the language, such as currently exists in the Mesa debugger.

(L11) Statically checked type system

(L12) Self-typing data (a la Lisp and Smalltalk), run-time type system

The primary value of type systems is descriptive and structural -- specifying the intended properties of one's data, and providing a mechanism for ensuring consistency between the suppliers and clients of an interface. Enabling a compiler to generate better code is secondary. There are several dimensions of variability in type systems:

> Whether type information is bound early, as in Mesa, or late, as in Lisp and Smalltalk. There is a need to provide the programmer a selection from a spectrum of alternatives -- Mesa provides variant records, which are a limited form of run-time binding, and Lisp has DeclTran, which provides some compile-time binding.
>
> Whether types form a strict partition of the data values, a coercion hierarchy, or some even richer structure. We believe that a richer structure is desirable. It might include provision for generic operators, for type-parameterized programs (Mitchell & Wegbreit "schemes"), for a more general notion of type as simply a partial specification of the behavior of an object (perhaps like the Alphard "needs" list), and for automatic pointwise extension of operators over collections.

It appears feasible to deduce much of the type information for a program automatically, starting from the assumption that each variable only takes on values of one type [Milner, Cousot & Cousot]. This would alleviate some of the nuisance of having to write type declarations. Name conventions such as those of the Software Production Environment, if interpreted by the compiler, would also eliminate most of the need for separate specification of type information.

This is an area ripe for research. We believe that, at a minimum, both the early-bound Mesa type system and the late-bound Lisp/Smalltalk type system must be supported (as alternatives) by an EPE.

(L13) Encapsulation/protection mechanisms (scopes, classes, import/export rules)

(L14) Abstraction mechanisms; explicit notion of "interface"

Abstraction mechanisms are important because they make explicit the logical dependencies of one part of a program on another, while concealing implementation choices irrelevant to the communication between such parts. Thus, these mechanisms enable the ability to factor the development, debugging, testing, documentation, understanding, and maintenance of programs into manageable pieces, while leaving individual programmers the appropriate freedom to design those pieces.

The ability to specify interfaces in the abstract, and to conceal their implementation, is important, but a difficult research area. It is possible to derive this information about implicit interfaces after the fact using tools like Masterscope. The code produced by an optimizing compiler need not reflect source-level modularization, if tighter binding improves efficiency and the user is willing to pay the price of more compilations and possibly decreased debugging information.

We believe one facility in this area is absolutely essential: it must be possible for a programmer to control which names get exported from a package. In addition, it is important for the system to conceal the distinction between user-defined packages and system-provided primitives (types, operations, etc.) at least as well as Mesa does.

(L15) Non-hierarchical control (coroutines, backtracking)

Coroutines and generators are essential: they provide a natural way to write transducers (programs that consume a data stream and produce another), which in turn are often the best way to modularize a data transformation algorithm. Lisp has backtracking, a control discipline sometimes used to explore alternatives in goal-directed searching, but there is considerable disagreement over the mechanism used to provide it, and its uses can probably be covered by more restricted coroutine and Undo mechanisms. Closures are a method of providing a wide variety of interesting control and binding environments, but we are not sure whether they can be implemented efficiently enough, or are structured enough, to replace the more specialized coroutine constructs.

(L16) Adequate runtime efficiency

The ultimate efficiency criterion is whether a system can meet its external specifications/constraints (responsiveness to human users or program clients). Computational efficiency equivalent to Alto Mesa, coupled with the larger real memory and faster disk of the Dorado, is adequate for many projects [we think we can attain this through internal re-engineering of Lisp and Smalltalk]; for other, more computation-intensive systems, at least another factor of 5 is attainable (based on raw hardware speed).

(L17) Inter-language communication

We believe the best way to attain the benefits of uniform methods for accessing the machine's facilities at the lowest level is for all language systems to run under a single operating system that is carefully constructed not to impose unnecessary space or time penalties on its clients. For the D0 and Dorado, we believe that Pilot satisfies this criterion.

Inter-language communication at a higher level helps reduce duplication of effort and also provides one way of attaining extra efficiency for particular functions. Calling Mesa subroutines from Lisp or Smalltalk will probably be adequate. Communication through the network or file system interface requires very little work, but is too inefficient. The general problem seems difficult and less important.

(L18) Uniform screen management

Use of the display is pervasive in our interactive systems. Lack of uniformity leads to duplicated effort, often of low quality since an individual builder cannot easily draw on all past experience or devote the time to taking advantage of it. On the other hand, too much central control over screen management may frustrate the desire to experiment with new paradigms for interaction.

We believe that it is possible to "virtualize" the screen and the user input devices -- that is, require people to write programs on the assumption that they will only have access to a subpart of the screen and to a slightly filtered stream of input events -- in a way that will not markedly impede our ability to experiment, and that will have a large payoff in terms of the user's ability to construct a screen environment containing multiple windows on different programs. At a minimum, the user must have direct access to all the capabilities of BitBlt, appropriately mapped or confined to work on a virtual screen.

(L19) Inheritance/defaulting (Smalltalk subclassing)

Languages that provide for programmer-controlled defaulting or inheritance reduce the time and chance for error in the programming process by making it unnecessary to write the same code or parameter values over and over again. The basic idea is that one should be able to write programs in a way that only specifies how they differ from some previously written program. Examples include default standard values for procedure arguments (how does this call differ from a "standard" call?), variant records (how does this particular record distinguish itself from the invariant part?), and the Smalltalk subclass concept (how does this class of objects differ from some more general class?).

We did not discuss this area beyond observing that it is somewhat related to the schemes question discussed under (L12), and that Smalltalk seems to derive considerable benefit from it.

(L20) Ability to extend language (e.g. operator overloading)

Languages may be extended by users in a variety of ways. Data structure extension, through user-defined data types and associated operations, makes it possible to write programs in terms of concept-oriented rather than implementation-oriented data objects. Syntax extension, through user definition of new language constructs, allows the user to define specialized notations that may be valuable for particular tasks: this is discussed in detail in (L21). Operator extension, the ability to define meanings for basic language constructs such as arithmetic or iteration when applied to user-defined objects, brings some of the benefits of notational extension with less drastic consequences in program readability.

Data structure extension is accepted as an important part of all modern language. Syntax extension has fallen into disfavor because of a lot of bad experience: we believe this happened partly because the tools did not support extensions as well as they did the base language. Operator extension is already present in a number of languages such as Algol 68: we did not discuss its merits. It is interesting to note that Smalltalk is founded on the notions of data structure and operator extension, but that the syntax extension facilities present in the 1972 version of the language have been removed.

(L21) Ability to create fully integrated local sublanguages

All language systems actually have many small sublanguages for special purposes. For example, Mesa has not only the Mesa language, but the C/Mesa configuration language, the debugger command language, the programming language subset acceptable to the debugger's interpreter, and the very small languages used to control the compiler and binder from the Executive command line. "Fully integrated", as an ideal, means that control and data should be able to pass freely between

sublanguages, and that the facilities (editor, prettyprinter, I/O system, etc.) applicable to the primary programming language should also be applicable to the other sublanguages.

Lisp is unique in that its dozen or so sublanguages all provide the ability to embed arbitrary Lisp computations in them, e.g. in the middle of the editor one can compute (by calling an arbitrary program) data to be inserted, possibly as a function of the thing being edited, or even a sequence of edit commands to execute. The following features of Lisp seem to have made the creation of integrated sublanguages easier:

> S-expressions are a simple, standard internal representation of parse trees for all sublanguages.
>
> Lisp provides a standard method of sharing names and passing environments, namely a single, very simple name environment (atoms) that all sublanguages share. (This has both advantages and drawbacks: it leads to the "FLG" phenomenon, for example.)
>
> Many internal system "hooks" are available to the sublanguage implementor. (This too has its drawbacks: it tends to make sublanguages more fragile.)
>
> The standard system contains packages (prettyprinter, table-driven lexical scanner and parenthesis parser) which make I/O of program-like structures easy.

The sublanguages that don't take advantage of these characteristics, such as CLISP, QLISP, and KRL, find their lives a lot more difficult. If we were willing to limit the complexity of sublanguages to that of S-expressions, i.e. procedures and conditionals, then we could devise an S-expression-like representation for Mesa also. (Extending this to, say, arithmetic expressions not only involves a complex parser and prettyprinter, but in a statically typed language like Mesa also requires taking type declarations into account to decide what the operators in the source text actually mean. Admittedly, the S-expression-like approach doesn't allow embedding of a reasonable subset of Mesa itself in a sublanguage, and it doesn't address the point that some of the highest payoff comes from the integration of languages, like KRL, that *don't* look like S-expressions.) We also noted that no matter what features the language and environment provided, proper proceduralization of facilities was essential: even Lisp has sublanguages -- in particular, the compiler control sublanguage -- that are implemented so as to interact with the user directly, and that therefore cannot be considered integrated.

To sharpen our ideas about what integration means, we considered a "straw man": a system in which all languages (editor, interpreter, ...) shared a screen interface (window manager) but were otherwise entirely separate. This led us to the following observations:

> This model was proposed as one that imposed minimal requirements on the individual subsystems, at least if they only dealt with the screen as a sequential character I/O device. This may not be a proper assumption, however: despite numerous attempts, no such package has ever been developed for the Alto, and this may be because no one has been able to develop a satisfactory model for the interface. We agreed that things become more complex as the subsystem's view of the display becomes more sophisticated (e.g. an editable document, a bitmap).
>
> While this model allowed for considerable communication (by human transfer of characters from output in one window to input in another), it had two serious deficiencies: it made no provisions for communication under program, rather than manual, control, and it required that all transmitted information be in string form. (Note that even transmission of file names requires integration in the sense of sharing a common file system.)

While we did not reach any clear conclusions, we were able to agree on the following:

> This is an important area for discussion, but it needs more time than we had available to us.
>
> Integration really means a common model for communication.
>
> The one catalog entry we now have should be broken down into multiple features of differing priorities.

If all the other priority A features in the catalog were provided, Mesa could readily support sublanguages of the complexity of S-expressions.

Adequate proceduralization is necessary for a package implementing a sublanguage to be usable.

A major source of difficulty is the sharing or passing of environment information between sublanguages. One part of this difficulty is simply addressing or naming objects to be shared. Another part is making sure that shared objects are interpreted the same way (in Mesa, making sure the communicants share the same declarations for the objects). One way around this is to have a limited number of globally agreed-upon structures, such as strings or S-expressions, and then encoding more specialized languages within them and interpreting them by convention or agreement (as Lisp does).

We also do not agree on the importance of fully integrated sublanguages: a number of Lisp users feel this item should have very high priority.

(L22) User access to the machine's capability for multi-precision arithmetic

Many language systems, though implemented on machines in which multi-precision arithmetic in assembly language is relatively straightforward, make it impossible for the user to get at these facilities (such as the carry from single-precision addition, or the double-by-single division that most machines provide in hardware). There is no excuse for this, especially on machines with relatively short (16-bit) words.

(L23) Good facilities for processes, monitors, interrupts

Synchronization between logically asynchronous processes is necessary in many programs, either for functional reasons (a system should be able to listen for incoming mail, send a file to be printed, and carry out interactive editing simultaneously) or for efficiency (overlapping computation with disk transfers). The language system, through an underlying operating system if necessary, should provide mechanisms that help the programmer write programs that involve multiple processes. There are a number of adequate, though conflicting, models of these mechanisms available, such as the Mesa and Pilot facilities and the Smalltalk scheduler. We did not discuss this area at all.

(L24) Simple, unambiguous syntax (including infix notation)

While CLISP leaves a very large gap between what can be precisely defined and what the user can reasonably do, and while Mesa syntax is regrettably complex (5 closely-spaced pages), we are willing to tolerate problems of this sort under the assumption that the primary users of the system will be experts, and that novices will be able to learn a useful subset easily.

(L25) Control over importation of names

As discussed under (L13) and (L14), import control seems less important than export control. The reason is that the implicit use of an interface can be deduced locally by nothing what names are used; for export, the issue is global and not under control of the provider of the package without some explicit specification.

(L26) User packages as "first-class citizens"

We would like user-defined packages to function as "first-class citizens" on a par with built-in primitives (types, operations, etc.). However, the Euclid experience seems to indicate that this is

very difficult. In Smalltalk, this goal has been achieved except for some I/O issues, at the expense of not having static type structure in the language at all.

(L27) Closures

See (L15) for discussion.

(L28) Full-scale inter-language communication

(L29) User microprogramming

The ability to share data and pass control freely between programs written in any of the major languages depends on carefully coordinated use of certain basic resources such as the peripheral devices and the machine's address space. We think this is less urgent than the more restricted facilities of (L17): the primary motivation for changing languages in mid-program is increased efficiency, and dropping into Mesa from Lisp or Smalltalk provides this, although it does not address the secondary motivation, which is the ability to take advantage of work already done (or more conveniently done) in another language.

In cases requiring extreme speed, it may be necessary to give the programmer a way to write application-dependent microcode and link it to the system in a way that provides at least some checking that it will not destroy the rest of the system. We did not discuss this.

(L30) Clean data and control trapping mechanisms

There was a long and inconclusive discussion. Apparently we don't really know what this point is about. At one extreme of "data trapping" there is a simple address trap, as on early machines. At the other extreme is KRL. No one was willing to espouse either extreme. We noted that

> programming with data abstractions can help with this problem, since there is then a better handle on when data is being changed.
>
> checking some predicate at periodic intervals (e.g. on every control transfer) may be quite adequate when data trapping is being used to catch "core smashing" types of bugs. Mesa already has this facility.
>
> many interesting cases can probably be handled by using the primitive trapping facilities of the mapping hardware.

(L31) "Good" exceptional condition handling

As discussed above, we really don't know what this would mean. Roy Levin's thesis, among other published papers, may be relevant.

TOOLS

(T1) Fast turnaround for minor program changes (< 5 sec)

Our concern with fast turnaround comes from the observation that programming should be "think bound", not "compute bound". There are several "knees" (points of substantial non-linearity) in one's perception of response delays. One such knee is in the vicinity of 3-5 seconds. We believe that it is essential to reduce the system time for minor program changes to below this point.

(T2) Compiler/interpreter available with low overhead at run time

The issues here are similar to those in (L9-10), namely, to reduce the mental and execution "gear-shifting" overhead caused by artificial divisions between compilation and execution environments. A sufficiently fast compiler is just as good as an interpreter for executing typed-in programs or programs constructed on-the-fly by other programs. However, it is essential that one be able to save some form of compiled code, for applications that embed procedures in data structures.

(T3) Cross-reference/annotation capability

(T4) Prettyprinter

These capabilities contribute substantially to the readability of programs, which in turn has a large effect on the ease of maintenance. A simple "batch" cross-reference facility is essential; more sophisticated facilities, such as those of Masterscope, are less urgent.

(T5) Consistent compilation

(T6) Version control

Consistent compilation is an efficiency issue: to get the right thing to happen without blindly recompiling and reloading everything. Version control is more fundamental. It has two major aspects: history and parameterization.

Under history, we want to be able to tell exactly how a particular system or component was constructed, and what it depends on (e.g. microcode version, interface definition or whatever). Furthermore, we want to be able to reconstruct a component automatically. This requires that every computation involved in its original construction must record *all* its inputs, and be prepared to repeat itself from this record. Since the inputs may be (references to) files, it is also necessary to have a naming scheme for files that is unique over the whole universe, and a guarantee that no file will ever be destroyed (unless the rule for reconstructing it is saved, together with all the required inputs).

Under parameterization, we want a systematic way of specifying the construction of a system that never existed before (e.g. it is for a new microcode version, or different implementations of the same interfaces are combined in a new way). We agreed that we don't aspire to solve this problem in the full generality required by IBM.

Replacing code in an existing system is in principle a special case of consistent compilation -- the general question is when a complete but expensive procedure (recompilation, reloading, etc) can be bypassed. We note that replacing code is in practice *not* just an efficiency issue, since getting the system back to the exact current state is not possible in general. The reason is that the current state depends on user program execution, and the user program cannot be counted on to follow the rules mentioned under history, which we impose on the system programs that make up the environment.

(T7) Librarian, program-oriented filing system (including Browser)

Coordinating access by multiple maintainers to a program made up of many packages requires some automation to avoid loss of consistency or even valuable information. DeSoto takes care of updating a "Bible" version from individual versions, and vice versa; the Mesa Librarian allows programmers to "check out" (in the sense of a library book) a module for modification. A system richly endowed with packages also needs some automation to catalog them and their documentation in a way that actively aids users in finding what they need.

(T8) Source-language debugger

It is essential that the programmer be able to debug using the same language constructs and concepts used in writing the original program. This is facilitated by a minimum of distinctions between "compile-time" and "run-time" environments -- see also (L10) and (T2).

(T9) Dynamic measurement facilities

These facilities are necessary to understand the behavior of complex programs under conditions of actual use. Smalltalk and Mesa have a Spy, which works by sampling the PC, and Lisp has Breakdown. The Smalltalk and Mesa facilities are relatively little used: the Mesa facilities are poorly documented and supported, and the nature of Smalltalk is such that there is often little meaningful tuning one can do. Proposed for Mesa by SDD are

> the Diamond Test Module's facility for counting frequency and time between any pair of breakpoints,
>
> a facility for writing an event in a log, either by procedure calls, or by action to be taken at a breakpoint, and
>
> a "transfer trap" mechanism that logs data at every control transfer, together with some standard ways of reducing this data to produce the same kind of information that a Spy produces.

We agree that something as good as Breakdown is good enough.

(T10) Checkpoint, establishing a protected environment

Checkpointing is needed to be able to protect oneself against unexpected machine or system failures. The weakness of the facilities in all three current systems was noted: file state is not saved, nor is there any check that it hasn't changed on a restart. "Protected environment" means the ability to install a new version of something in a system and still be able to revert to the old version very cheaply; cheap checkpoints can provide this. Cheap checkpoints can be done in a paged system with copy-on-write techniques.

(T11) History and undoing

"History" refers to the system keeping a typescript file, the ability to feed information from the typescript back to the system, and the ability to have a handle on the values returned as well. This is an attribute of the interactive interface: its value comes from the observation that the operations one performs often are similar to, or use the results of, the operations one has recently performed.

An "undoing" mechanism should cover both system-implemented actions such as edits, and a way for users to supply a procedure that will undo the effects of another specified procedure. This can

be a very inexpensive alternative to checkpointing as a way to give the user the ability to experiment with alternatives without imposing the burden of manually saving and restoring the relevant state.

(T12) Editor integrated with language system

Editing is just one function of a language system, carried out using a particular sublanguage. (In fact, we believe it may be the appropriate model for the *major* sublanguage presented to the user at the terminal.) As such, it should be integrated with the rest of the language system in that

> the user doing editing can call on arbitrary programs to compute commands or data needed for the editing process, including the ability to pass selections from the thing being edited to the computation as arguments;
>
> any program can call on the editor as a package.

The latter seems very useful and relatively easy to achieve. We agree the former is also valuable, but there is disagreement over whether it is merely valuable or extremely important.

(T13) More optimizing compiler if user willing to bind more tightly -- with full compatibility

A recurring source of difficulty in programming is the generality/efficiency tradeoff: when it is time to tune a system for better performance, it is traditionally necessary to make major logic changes as well. An alternative is to keep the logical structure of the program the same, but have a compiler that can do the necessary rearrangements as part of the compilation process: the programmer instructs it as to what kinds of flexibility or modularity should be sacrificed, and the critical choices to be made in the representation of data structures. The Lisp "block compiler" is one example of this idea; the in-line procedure feature suggested for Mesa is another. We did not discuss this area except in passing.

(T14) Aids for incremental development (stubs, outstanding task list)

Top-down programming, or independent development of modules in a system, often benefits from the ability to replace as yet unavailable modules with "stubs" which have the same functional behavior but simpler (and presumably less efficient) implementation. This and other aids for keeping track of the status of parts of a large project have been used successfully in many system development efforts.

(T15) Regression testing system

(T16) Random testing aids

Both regression testing -- keeping a record of standard tests and results with a program module, and automatically checking them after a change to the module -- and testing with random data have proven to be worthwhile methods for checking programs too large or complex to verify or describe analytically.

(T17) (high capability) Masterscope

Any facility like Masterscope, which maintains an up-to-date data base of relations among parts of a program, must be integrated into the system at a fundamental level. Our discussion revealed that the fundamental aspect is the need for a *single* funnel for changes to the system (Mesa pretty much has this now, but Lisp does not). Relation to the file system was discussed, and it was agreed that manual use of the file system should be outlawed. A consequence is that the PE must do recovery at least as well as the file system does. Of course, having a reliable file system underneath makes

this much easier. A variety of techniques are possible, which we did not explore in detail.

(T18) Access to on-line documentation (Helpsys)

Good on-line documentation, both for reference and for learning, can greatly reduce the need for time spent studying an enormous manual, can provide instant cross-linking of related subjects in a way that hardcopy cannot, and can use one's current context to implicitly locate relevant material -- Lisp's Helpsys facility is unique in these respects. However, creating and maintaining such documentation is a tremendous amount of work, even if the process is partly automated.

(T19) Static analyzers: verifier, performance predictor

We agree that, especially for programs used by many people in low-tolerance environments, an ounce of prevention is worth a pound of cure: effort expended on eliminating bugs or bottlenecks beforehand can save a lot of time and trouble locating them afterward. Unfortunately, verification technology is still unable to accommodate programs of significant size written in languages of realistic complexity, and very little has been done on deriving performance information from the program text (in contrast to analytic models of systems at a gross level, of which there are many).

PACKAGES

The contents of this section were largely extracted from Butler Lampson's memo of March 9, 1978 titled "Components for office systems" (file [Maxcl]<Lampson>components.memo). The interested reader is referred to that memo for additional discussion of the items below and of the role of packages in a programming environment. There is a lot of overlap between this section and the previous ones on VM/Language and Tools, since many of these packages will be required to implement the facilities mentioned in those sections.

Beyond the virtual machine and programming language, which are forced on all users, and the tools, which should be applicable to all users, the quality of a programming environment is largely determined by the presence of packages that provide functional capabilities useful to many applications. We cannot stress too strongly that, from our experience, the only way to ensure the necessary high quality for such packages is to have a very small group (1-4 people) with the final authority and responsibility for deciding which packages are to be incorporated in the system in a way that makes them readily available to all.

(P1) Text objects and images

(P2) Line objects and images

(P3) Scanned (bitmap) objects and images

(P4) Press files

The manipulation of images is of primary concern to us in our experimental systems. We can divide these manipulations into two categories:

> Manipulation of abstract objects such as formatted documents, forms, line drawings, pictures in AIS or related forms. The operations on these objects are defined by the semantics of the objects, not by their representation on a medium.
>
> Manipulation of the images of these objects on displays or printers. These operations must take the nature of the medium into account.

In the first category we might find editing operations on document files such as insert, replace, search, etc.; on drawings and pictures such as scale, rotate, reflect, clip, shade, connect points with a spline curve, etc. In the second we find operations for mapping objects onto media in a variety of ways, some of which must be reversible (e.g. when a user makes a selection in the displayed image of a document, that selection really refers to the data in the document itself).

We believe that enough experience has been gained in these areas that it is possible to construct packages that will be useful in a wide range of programs, and that will markedly decrease the effort required to write programs that use them.

An interesting area that we have not discussed per se is the general notion of "annotation" of documents (or data structures): formatting information can be considered an annotation to the text, comments to a program, meta-descriptions in the KRL sense to a slot or another description. Pilot, for example, provides a notion of "subsequences" of a byte stream, which can easily be used to represent formatting information in a way that uninterested programs can ignore.

(P5) More elaborate screen management

In addition to the basic screen management capabilities mentioned under VM/PL above, there are some additional facilities (scrollable windows, for example) that many programs will want to share. Again, we believe there will be a payoff from the presence of some carefully designed packages in the environment.

(P6) Remote file storage

The manual transfer of files between machines is a significant source of errors and wasted time. Such transfers are necessary either because of space problems or because one machine has a capability (such as a printer or high-performance display) not possessed by all. Pilot provides the ability for any program to access files stored on a remote server as though they were stored on the local disk.

(P7) Small data base manager

As a goal, we believe that the well-integrated access to large data bases mentioned under VM/PL has a potentially enormous payoff, since many tools as well as experimental programs will benefit from it. However, if it turns out that we can't figure out how to provide this, then we will need a well-designed data base package for managing locally stored data.

(P8) Message transmission system

Message transmission is a useful paradigm for many kinds of inter-machine communication.

(P9) Remote procedure call

The ability to call a procedure on another machine as though it were on one's local machine is a different, less well understood communication paradigm.

(P10) Event logging

Event logging is a useful technique for redundancy and crash protection, for gathering statistics, and for reducing the cost of updating a data base in response to events affecting it.

(P11) Background processing

In an interactive system with enough real memory, both external communication (sending and receiving mail, printing) and computation (recompilation, Masterscope data base maintenance) can make effective use of time when the user is thinking.

(P12) Generalized cache

Many applications can benefit from a cache mechanism that provides local copies of more remote data, e.g. copies in memory of data from the disk, or copies on a local machine of files stored remotely. A package could keep track of which items had been used least recently, schedule rewriting of changed items, and deal with locks and timeouts.

(P13) Document editing

Laurel had to re-implement a modest but significant part of the editing capabilities of Bravo. There is no good reason why anyone should have to do this again.

(P14) Forms

(P15) Menus & other standard user interfaces

Packages that provide standard user interface tools such as forms, menus, selection, etc. are desirable both in the interests of uniformity and simply to save work.

(P16) History lists

Programs should be able to take advantage of the same mechanisms used by the system to provide the history and undoing capabilities discussed in (T11).

(P17) User access to full bandwidth of disk

Data base manipulations and code overlaying require brief bursts of high-bandwidth disk activity. The system should not prevent the programmer from using the disk's full bandwidth, and a package should make it easy (like the ScanFile package for the Alto).

(P18) (English) dictionary service

Office applications involving documents can benefit from easy access to an English dictionary (for spelling correction, hyphenation, and thesaurus applications, for example).

(P19) Teleconferencing

Inter-person communcation should play more of a role in our future experiments; we need a package to handle the mechanics of keeping several users' views of the screen, cursor, etc. consistent.

(P20) Audio

We have hardware support for capturing and playing back audio information, but hardly any software support. Something like the current audio message system ought to be a very small project.

(P21) User access to full bandwidth of networks

As in (P17), the system should not obstruct the programmer's access to the machine's full I/O bandwidth in experimental situations.

OTHER

(X1) Adequate reference documentation

Reference documentation must be complete and reasonably well organized and indexed. The Interlisp manual is a shining example of how well an entire environment can be documented. It also demonstrates that keeping this documentation up to date is a lot of work.

(X2) CSL control over the system's future

To the extent that we build our programming environment on a system being developed and maintained by another organization (e.g. Mesa and Smalltalk), we run the risk of having our concerns short-changed. Conversely, if we build on a system we are supporting for outside users (e.g. Lisp), we will continually need to balance our own priorities against those of the outside community, and the need to remain compatible.

(X3) "Efficient" interface for experts

For experts, the desire for common operations to require a minimum of human effort often rightly takes precedence over the desire for the greatest possible uniformity or simplicity in the human interface. However, such interfaces are too often constructed without paying any attention to the few principles of interface design that we do know. We believe it is important to consider consciously the design of certain key command interfaces (editing, debugging, screen management).

(X4) Uniformity in command interface

(The following paragraphs were adapted from the user guide for the OIS Tools Environment, written by SDD.)

In the process of using interactive programs, such as the tools listed in the previous major section, the user will inevitably accumulate perceptions, opinions, and models of the programs, and conjectures as to their workings. The net effect of these perceptions is referred to as the "user illusion". The intent is to allow the user to see the programs only in relation to his own needs and purposes, and not have to concern himself with the internal workings of the programs. What is important about a standard user interface package is that the user be able to confidently predict the general manner of interaction with a program that uses the package, even though he hasn't experienced it yet; and that, by and large, the user will be right. This has been called the Law of Least Astonishment.

The concept of a consistent user interface also simplifies the design and coding of any program that interacts with the user. By adopting the conventions and making use of the facilities, it is a relatively simple matter to create useful interactive programs, because the programmer can concern himself with the algorithm rather than with creating his own user interface.

We believe that, in addition to consistency, another important goal to pursue in the design of the user interface might be called the Principle of Non-Preemption. Individual interactive programs should operate in a non-intrusive manner with respect to the user's activities. The system does not usurp the attention and prerogatives of the user. A program responds to the user's stimuli, but then quietly retains its context and logical state until the user elects to interact with the program again, not (for example) monopolizing the resources of the computer.

(X5) "Self-teaching" interface for beginners

(X6) Good introductory documentation

Since we are concerned with a programming environment primarily for CSL, and secondarily for the rest of the local research community, we feel that concern for novices should have low priority, since the rate at which new people join the community is low and most of them are already sophisticated.

Priority ranking and interrelation of capabilities

We arrived at a priority ranking for capabilities by giving each member of the working group 100 votes to be divided among the capabilities on the list. The vote total below is simply the sum of the votes of the 5 members who actually voted, identified by their initials: D(eutsch), H(orning), L(ampson), M(orris), and S(atterthwaite). (As a check, we also ranked the capabilities according to the median rather than the total, and the results were essentially the same.) We found that a natural division into 5 priority groups emerged from the ranking. The reader should be aware that we were ranking capabilities on their utility if present in some reasonable form, not on the value of doing research into how to improve what we now know about providing them.

We were able to reach a consensus about how fundamental each capability was, in the sense of how difficult it would be to add that capability if it was not allowed for in the initial system design. In doing this we drew heavily on the experience gained from the three existing language systems. This consensus is expressed below according to the following code:

F - Fundamental, much harder if not allowed for originally

I - Intermediate, somewhat harder if not allowed for

A - Add-on, difficulty does not depend significantly on pre-planning (although it may be intrinsically hard anyway)

We identified an "enabling" relation between certain pairs of capabilities, in the sense that capability X is almost certainly required to provide capability Y. This relation is expressed below by an arrow "<-" meaning "is enabled by". So many things depend on item (L1) that we have omitted mention of this.

Finally, we estimated the difficulty of providing each capability in each of the three presently existing environments (Lisp, Mesa, and Smalltalk). Each capability has a 3-digit difficulty code referring to the effort required to provide the capability in Lisp, Mesa, and Smalltalk in that order. The difficulty codes have the following meanings:

0 - available
1 - easy
2 - straightforward but takes time
3 - hard
4 - impossible (we found we didn't need to use this)

PRIORITY RANKING

In the tabulation of votes, "x" means zero votes, while "*" means that the person gave no votes to this item because he assumed it would be provided anyway. Votes of the form "3/4" mean 3 votes if we are primarily interested in E(PE), i.e. investigation of programming environments per se, but 4 votes if we are concentrating on (EP)E, i.e. production of experimental programs.

Priority A

			D	H	L	M	S	Total	
(L5)	F	030	6	x	10	20	4	40	Object management -- garbage collection/reference counting
(L11)	F	203	3	6	9	7	5	30	Statically checked type system
(L4)	I	000	6	7	*	10	4	27	Memory management -- object/page swapping
(L14)	F	302	2	4	5	10	4	25	Abstraction mechanisms; explicit notion of "interface"

(T1)	I	020	3	x	8	5	4	20	Fast turnaround for minor program changes (<5 sec)
(L16)	F	?0?	3/4	6	2	4	3	18/19	Adequate runtime efficiency
	<- (L17) Inter-language communication								
(L1)	F	002	6	7	*	x	5	18	Large virtual address space (≥ 24 bits)

Priority B

(L13)	F	302	3	5	2	4	3	17	Encapsulation/protection mechanisms (scopes, classes, import/export rules)
(L3)	F	333	3	4	8	x	x	15	Well-integrated access to large, robust data bases
(L12)	I	020	4	3	3	x	5	15	Self-typing data (a la Lisp and Smalltalk), run-time type system
(T5)	A	203	3	2	3	5	2	15	Consistent compilation
(T6)	I	323	2	x	8	3	2	15	Version control
(T8)	F	000	4	1	*	5	5	15	Source-language debugger
	<- (L9) Program-manipulable representation of programs								
	<- (L10) Run-time availability of source program information								
(P1)	A	121	2	3	3	4	3	15	Text objects and images
(L18)	I	021	2	4	2	4	3	15	Uniform screen management
(L8)	A	202	2/4	3	2	4	3	14/16	User access to the machine's capability for packed data
(L10)	F	111	3/2	3	4	x	4	14/13	Run-time availability of all information derivable from source program (e.g. names, types, scopes)
(X2)	F	222	3	5	5	x	x	13	CSL control over the system's future

Priority C

(L2)	A	102	x	6	x	x	4	10	Direct addressing for files (segmenting)
(L6)	I	000	2	5	x	2	1	10	Some support for interrupts
(T2)	I	021	3/2	1	1	2	2	9/8	Compiler/interpreter available with low overhead at run time
(X1)	I	012	2	4	x	x	3	9	Adequate reference documentation
(T7)	A	222	2	1	1	2	2	8	Librarian, program-oriented filing system (incl. Browser)
(L9)	I	012	3/1	3	x	x	2	8/6	Program-manipulable representation of programs
(T9)	I	011	2/3	1	x	2	2	7/8	Dynamic measurement facilities
(P3)	A	222	2	2	1	2	x	7	Scanned (bitmap) objects and images
(P4)	A	120	1	2	1	x	3	7	Press files
(X3)	F	010?	2	1	3	x	1	7	"Efficient" interface for experts
(P2)	A	222	1	2	2	x	1	6	Line objects and images
(P6)	A	001	1	x	3	x	2	6	Remote file storage

Priority D

(L17)	I	202	1	4	x	x	x	5	Inter-language communication

(T11) F 122 x 2 3 x x 5 History and undoing

(L15) F 000 2 x x x 2 4 Non-hierarchical control (coroutines, backtracking)

(L20) F 220 x x 4 x x 4 Ability to extend language (e.g. operator overloading)

(L21) I 333 2/1 x x x 2 4/3 Ability to create fully integrated local sublanguages

<- (L9) Program-manipulable representation of programs
<- (L10) Run-time availability of source program information
<- (L12) Self-typing data

(L27) F 121 1 x x 3 x 4 Closures

(T10) I 000 1 x 2 x 1 4 Checkpoint, establishing a protected environment

(L19) F 321 x x 3 x x 3 Inheritance/defaulting (Smalltalk subclassing) (difficulty depends a lot on how much it has to do)

<- (L13) Encapsulation/protection mechanisms

(T3) A 111 2 x x x 1 3 Cross-reference/annotation capability

(T4) A 011 1 x 2 x x 3 Prettyprinter

<- (L9) Program-manipulable representation of programs

(P15) A 111 2 x x x 1 3 Menus & other standard user interfaces

(P13) A 020 2 x x x 1 3 Document editing

(L7) F 112 x x x x 2 2 Adequate exceptional condition handling

(T12) I 021 2 x x x x 2 Editor integrated with language system

(P9) I 222 1 x x x 1 2 Remote procedure call

(T13) F 003 x x x x 1 1 More optimizing compiler if user willing to bind more tightly -- with full compatibility

(T18) A 011 1 x x x x 1 Access to on-line documentation (Helpsys)

(P8) A 212 x x x x 1 1 Message transmission system

(P10) I 111 x x x x 1 1 Event logging

(P12) A 222 x x x x 1 1 Generalized cache

(P14) A 222 1 x x x x 1 Forms

(X4) F 222 1 x x x x 1 Uniformity in command interface

Priority E (no votes at all)

VIRTUAL MACHINE / PROGRAMMING LANGUAGE

(L2b) ? 222 An enormous virtual address space ($\geq$ 48 bits)

(L22) A 100 User access to the machine's capability for multi-precision arithmetic

(L23) I 202 Good facilities for processes, monitors, interrupts

<- (L6) Some support for interrupts
<- (A15) Abstraction mechanisms

(L24) F 120 Simple, unambiguous syntax (including infix notation)

(L25) F 302 Control over importation of names

<- (A15) Abstraction mechanisms

(L26) F 331 User packages as "first-class citizens"

(L28) F 333 Full-scale inter-language communication

<- (L17) Inter-language communication

(L29) A 222 User microprogramming

(L30) ? 333 Clean data and control trapping mechanisms

<- (L6) Some support for interrupts
<- (L23) Good facilities for processes, monitors, interrupts
<- (L31) "Good" exceptional condition handling

(L31) ? 333 "Good" exceptional condition handling

<- (L7) Adequate exceptional condition handling
<- (L13) Encapsulation/protection mechanisms
<- (L14) Abstraction mechanisms
<- (L15) Non-hierarchical control structures

TOOLS

(T14) A 222 Aids for incremental development (stubs, outstanding task list)

<- (L9) Program-manipulable representation of programs

(T15) A 222? Regression testing system

<- (L9) Program-manipulable representation of programs

(T16) A 333? Random testing aids

(T17) F 022 (High capability) Masterscope

(T19) A 333 Static analyzers: verifier, performance predictor

<- (L9) Program-manipulable representation of programs

PACKAGES

(P5) A 022 More elaborate screen management

(P7) A 221 Small data base manager

(P11) F 323 Background processing

(P16) A 011 History lists

(P17) I 111 User access to full bandwidth of disk

(P18) A 222 (English) dictionary service

(P19) A 222 Teleconferencing

(P20) A 222 Audio

(P21) A 111 User access to full bandwidth of networks

OTHER

(X5) F 222 "Self-teaching" interface for beginners

(X6) I 222 "Good introductory documentation

Relation to existing environments (Lisp, Mesa, Smalltalk)

The relationship between our desired EPE capabilities and our existing language systems is complicated by the fact that there are two Lisp systems and two Smalltalk systems that must be taken into account. The present Lisp and Smalltalk systems have a kernel written in Bcpl and Alto assembly language respectively; they do most of their I/O directly. This implementation limits them to using 128K of memory and makes communication with the Mesa world on the Dorado very difficult.

It is intended that the Lisp kernel be converted from Bcpl to Mesa, and that Lisp's current I/O and paging facilities be replaced by those in Pilot. There is no current time schedule or manpower commitment for this effort. However, we believe that this step is unavoidable if Lisp is to serve as the base for an EPE. Accordingly, our difficulty estimates reflect the assumption that this has been done. The conversion process itself is somewhere between 1 and 2 in our coding scheme.

Smalltalk is in somewhat worse shape, since the current implementation uses an object swapping scheme and an addressing mechanism that do not scale up well. There is a design, roughly 85% complete, for a high-performance Smalltalk for the Dorado that uses paging and differs in many ways from the current Alto implementation: this design has been scrutinized by many people, but is not ready to implement. We believe that such a re-implementation is mandatory if Smalltalk is to serve as the base for an EPE: it has difficulty somewhere between 2 and 3. Again, our difficulty estimates assume that this basic step has been taken.

Mesa already has nearly all the machinery required to take advantage of the Dorado environment. The chief question mark at the moment is the availability of Pilot for the Dorado, which again we assume as a prerequisite. We are not aware that any manpower has been committed to this end. Bravo, the only editor currently used for Mesa programs, does not run well on Dorado and cannot be integrated into the Mesa/Pilot environment; we assume that some Mesa-implemented editor, perhaps Diamond or part of Laurel, will be available.

Discussion of Difficulty Estimates

In this section we discuss the difficulty estimates for the highest-priority features. Nearly all these features fall into one of three difficulty patterns:

> Considerably easier in Mesa than in either Lisp or Smalltalk (e.g. L11, L13, T5).
>
> Considerably easier in either Lisp or Smalltalk than in Mesa (e.g. L5, T1, L12).
>
> Comparable in difficulty in all three systems (e.g. L4, L3, T8).

This is not terribly surprising, given the underlying philosophical similarities between Lisp and Smalltalk.

Priority A

(L5) 030 Object management -- garbage collection/reference counting

Reference counting, which relies almost entirely on "local" information, and garbage collection, which needs a "global" map of all potentially traceable data, present quite different problems for Mesa. For reference counting, it might be adequate to add an attribute to the Mesa type system so that reference counting or transaction queuing can occur when the program stores into a pointer that might point to an automatically managed object, and arrange for a user-supplied finalization procedure to be called when a count becomes zero. A further refinement might be automatic generation of the procedure by the compiler based on declarations. All this would be difficulty level 2.

Garbage collection in Mesa is more difficult, for two reasons:

> The type information needed to locate all pointers is not present in the actual data, but must be derived from the symbol table produced by the compiler.
>
> The presence of LOOPHOLEs, pointer arithmetic, overlaid variants, and relative pointers makes it impossible for a garbage collector to find or trace all pointers to structure to be saved.

The former problem is only one of efficiency: one way around it might be to arrange for the compiler to generate a tracing procedure for each structure that might participate in garbage collection. The latter, however, is a fundamental difficulty. We believe that the proper approach to overcoming it is to discover a subset of the Mesa language that does not use any of the LOOPHOLE-like features mentioned above, and find a way to draw a protection boundary around a program written in this subset so that it can have automatically managed data. This is a hard problem.

(L11) 203 Statically checked type system

The basic problem with static type checking in both Lisp and Smalltalk is that very little is known at compile time about the referents of names.

Lisp currently has a facility called DeclTran, which uses embedded declarations (both of types and of arbitrary predicates) to check the types of operands at runtime and to generate more efficient code within an individual function. This facility applies to both interpreted and compiled code, but is somewhat machine-dependent and does not exist for the Alto/Dorado Lisp system. Also, it has no ability to deal with static checking of interfaces between user-defined functions (as opposed to the interface from user-defined functions to built-in operations like arithmetic). If mechanisms, such as those of (L13) and (L14), were added to Lisp to control exporting and linkage of names, it is likely that DeclTran could be extended to inter-function type checking. We are inclined to think that this is not a good approach in the long run and that the type system should be more tightly integrated with the underlying language system, but it seems like a workable way to obtain the desired result.

Smalltalk has no facilities at all for static declaration of types. Furthermore, because every operation is decoded at execution time by the object that is its first operand (even "elementary" operations like +), there is in principle no way of knowing what code will actually get executed in advance of execution time without some combination of type declarations and searching the entire system to discover what classes implement what operations. The situation is complicated by Smalltalk's subclass structure, which makes it very desirable that, for example, a method requiring an argument of type Number should accept values of type Integer (a subclass of Number). Smalltalk has a fairly simple compiler and name structure, but a lot of new machinery would have to be built, some of which would require careful thought about what "type" means in the Smalltalk environment.

(L4) 000 Memory management -- object/page swapping

Lisp, Mesa, and the high-performance Smalltalk all use a straightforward paging scheme, which Pilot supports.

(L14) 302 Abstraction mechanisms; explicit notion of "interface"

As discussed under (L13) below, Lisp currently has only a very weak mechanism for explicitly dealing with interfaces between a package and its clients. In addition, Lisp has no mechanism for taking a group of functions and controlling how its imported names are to be linked: such a facility is necessary if interfaces are to be separated from their implementations.

Smalltalk is somewhat better off since its invocation mechanism does hide the implementation from the caller. A type system for Smalltalk, as discussed under (L11) above, would also go far towards providing an explicit notion of interface, since the current Smalltalk notion of "abstract class" can be viewed as either a type or an interface description.

(T1) 020 Fast turnaround for minor program changes (< 5 sec)

Lisp has an interpreter, and a fully integrated editor. (We have discussed replacing the interpreter with a high-speed, low-code-quality compiler, which already exists for the Alto.) Smalltalk has a compiler which we believe will run fast enough on the Dorado, and an integrated editor.

Mesa offers several medium-size obstacles to fast turnaround for changes:

> Since the editor (Bravo) is not integrated or even properly packaged, considerable machine and human overhead is involved in getting in and out of it. We recommend replacing it by a packaged editor available in the programming environment. It appears that SDD is quite interested in this already.
>
> The compiler is not designed to compile anything smaller than an entire module. We recommend some work to modularize the compiler (it is already broken up quite well) and to make sure enough information is available for it to be able to compile single procedures.
>
> The current Mesa system does not provide for safe, incremental replacement of modules or procedures by new versions without losing execution state. This feature must be provided.

(L16) ?0? Adequate runtime efficiency

The Lisp instruction set is actually quite similar to that of Mesa: for example, it can use the Dorado IFU. Sources of additional overhead include the fact that all data are doublewords, and considerable runtime type-checking is required. We believe that even the present (Bcpl-based) Lisp system will perform adequately (better than 1x Maxc) on the Dorado; this will improve considerably after the move to Mesa and Pilot, which will enable use of the IFU. Further significant gains in efficiency would be possible with some redesign that would use the Dorado processor stack, and with the extensions required for (L11), static type checking.

The Smalltalk instruction set, in principle, is also similar to that of Lisp and Mesa. The high-performance Smalltalk design uses the IFU and processor stack. In addition to doubleword data and runtime type-checking, Smalltalk also suffers somewhat from substantially less ability to compile things like arithmetic and structure access in-line, and the requirement for doing some kind of hash lookup on each invocation. We believe that Smalltalk could also be brought up to acceptable efficiency through the high-performance design.

(L1) 002 Large virtual address space ($\geq$ 24 bits)

Lisp and Mesa already provide 24-bit address spaces. The difficulty code for Smalltalk reflects Dan Ingalls' estimate that the current system could be modified (without changing its basic structure) in 3 to 6 months.

Priority B

(L13) 302 Encapsulation/protection mechanisms -- export control

Lisp already has one mechanism for controlling the export of names from a group of functions: the block compiler. However, this mechanism has many disadvantages:

> Using it requires giving up a good deal of debugging capability, since non-exported function and variable names are completely lost.
>
> It is not compatible with either the interpreter or the normal compiler.
>
> Its application is limited to a single level, e.g. a package cannot use this mechanism to have subfunctions local to an individual interior function or group of functions.

We believe that adding export control to Lisp would require introducing notions of nested or otherwise controlled lexical name scopes similar to those of Mesa; we would expect this to have ramifications throughout the system, and some impact on user programs that take advantage of the fact that there is a single space of names that encompasses both all of the system, their own programs, and their own data. (We note that in the past, the lack of export control has often been seen as an advantage, since it made it easy for a somewhat knowledgeable user to extend or modify the system as desired, but we believe this is outweighed by the problems it causes.)

Smalltalk has two different export control mechanisms. For variables, there is a dual lexical hierarchy based on lifetime (method variables, instance variables, and static class variables) and on specialization (subclassing). For operations, each class defines the names of operations it will accept. However, Smalltalk is deficient in several respects:

> There is a loophole, fairly easy to use, that allows access from outside to any variable of any object.
>
> A class cannot restrict purely internal operations to internal use -- all operations within the class are accessible to all clients.
>
> Subclasses have access to all operations and data of their superclass.

We believe that remedying these defects would require significant additions to Smalltalk but no changes in its basic structure.

(L3) 333 Well-integrated access to large, robust data bases

This is an area requiring considerable thought. No general-purpose programming language currently offers this capability as we intend it to be interpreted. We have no a priori reason to think it would be harder to implement in one language than in another -- it will be a challenge in any of them.

(L12) 020 Self-typing data (a la Lisp and Smalltalk), run-time type system

Mesa, unlike Lisp and Smalltalk, segregates type information from data at runtime by relegating the former to symbol table structures put out by the compiler, which are normally not part of the runtime environment. Extending the type system into the runtime environment in a reasonable way seems to require adding types as a type in the language, and arranging things so that the runtime representation of types can match up properly with the information in the symbol table. Some of the design for this already exists, since the debugger must manipulate types internally. It is worth noting that Mesa's facilities for variant records and (proposed) for sequences already address a small part of this problem.

(T5) 203 Consistent compilation

Mesa ensures consistent compilation by placing time stamps on source and object files, and by recording in each object file the complete list of time stamps for the files that produced it. For Lisp and Smalltalk, the basic problem is that identified in (L13) and (L14) above, namely, lack of mechanisms for identifying the dependencies between modules.

(T6) 323 Version control

Mesa contains most of the mechanisms for the history part of version control (knowing exactly how to reconstruct any object file). Some things are missing, such as a record of the compiler switches (fixed in Mesa 4.1), and also the Binder does not provide this information in the files it constructs. For Lisp and Smalltalk, the problem is that objects do not carry any kind of identifying stamp, nor do compilation processes record the information they used. In the present Lisp system, the problem is almost insoluble, because compilation takes place in the context of whatever random collection of macros, flags, and other declarations happens to be lying around at the time of compilation. (The problem is not the large amount of state on which compilation depends, but the fact that there is no mechanism for recording the relevant parts of it with the compiled file.)

Aside from C/Mesa and the Binder, none of the three systems has any mechanism for describing how parts are put together to form a whole. C/Mesa fails to cover certain crucial aspects of initialization and parametrization -- there is no way to pass parameters from a C/Mesa program to initialization code, for example, in lieu of supplying them directly when the program initializes itself. Lisp allows programs to load other programs as part of their initialization, and several large systems have more elaborate mechanisms of their own for doing this, but there is no system-wide machinery of this kind.

(T8) 000 Source-language debugger

All three systems have adequate source-language debuggers. All of them could profit from additional work, since there are ideas in each one that are not present in the others: for example, the REVERT feature in Lisp, the ability to point directly to the source text on the screen in Mesa, and the Smalltalk method for displaying local variables.

(P1) 121 Text objects and images

Lisp already has a package (TXDT) for manipulating documents with formatting on individual characters (font, bold/italic, etc.). However, TXDT has no provisions for paragraph-level formatting or tables, and there is no software to convert to or from Bravo or Press files. These defects are minor. Facilities for manipulating text images as images are provided within DLISP. Documentation for the formatting aspects of TXDT, and for all of DLISP, does not currently exist, although the latter is being worked on.

No comparable package exists in Mesa, although Laurel and other systems have had to provide some of these functions. Diamond probably has an internal package that provides these capabilities, but it runs in a highly tailored environment (it doesn't even use the standard Mesa runtime system). When and if Diamond is converted to run under Pilot, this problem may disappear.

Smalltalk provides the notion of a formatted paragraph, and facilities for editing, displaying, and selecting within it, in the standard system. Smalltalk also provides conversion to and from Bravo format, and to Press format. Smalltalk does not provide paragraph-level formatting or tables, and its facilities for handling text images in any but the most straightforward way are poor. Again, these defects are relatively minor.

(L18) 021 Uniform screen management

Lisp already possesses an elaborate and functionally rich screen management package, DLISP. As noted above, DLISP is still being documented. It does seem to satisfy our principal desire, namely that programs access the display only through an interface that makes sharing and allocation of the physical screen invisible to them; we are not certain whether it imposes too much structure on the user's view of the display, to the extent that some kinds of experiments might be difficult or impossible (no such difficulties have arisen in the few applications attempted thus far).

Several attempts have been made at creating a screen management package in Mesa; all have failed to gain acceptance, for a variety of reasons. We believe that it will be necessary to start from scratch, taking into account the experience with the failed models and with the more successful ones of Lisp and Smalltalk.

Smalltalk has the opposite problem: accessing the screen in an anarchic way has been too easy. Some moderately successful interfaces have emerged -- one for windows, one for documents, and one under development for graphics -- but there is still a substantial need for unification of what has been learned.

(L8) 202 User access to the machine's capability for packed data

Lisp and Smalltalk, unlike Mesa, take the view that all data accessible to the programmer are pointers. Intrinsically non-pointer data, such as string characters, are handled with various subterfuges. To obtain the benefits of packed data within these two systems would require extending the storage management system to be able to allocate and deallocate such objects, and extending the compiler to generate instructions to access them under safe conditions. The Lisp "record package" accomplishes some of this, but in a way that is poorly integrated with the rest of the storage management system and makes use of what we consider impermissible loopholes in the compiler. Smalltalk has no such facility at all, although the Smalltalk implementors have discussed it and believe they understand in principle how to do it without structual changes in the system.

(L10) 111 Run-time availability of all information derivable from source program (e.g. names, types, scopes)

All three systems make most of the source information available in some form at runtime. In Lisp, the facilities for retrieving source information are very good, but they do not handle declarations and variables with the same ease as functions. In Mesa, the relevant data structures (in the symbol tables produced by the compiler) are not currently documented for public use. In Smalltalk, the information required to correlate the dynamic environment with the names used in the program exists but not in convenient form. Mesa's facilities for correlating the PC with a point in the source program are much better than either Lisp's or Smalltalk's.

(X2) 222 CSL control over the system's future

The basic problem here is that all three systems are currently enjoying a great deal of support by groups other than CSL, which however have priorities different from ours. We would have to provide this support if we were to choose any of these systems as an EPE base. The difficulty code reflects the investment in learning time and ongoing manpower required for such support.

CSL certainly has control over the Dorado Lisp system, and over the Lisp-coded parts of the Tenex implementation. However, our real control over Lisp's future is hampered by our desire to maintain

compatibility with a large number of outside Interlisp users so we can benefit from mutual exchange of programs. Against this we must weigh the benefits of tailoring Lisp to our high-performance, display- and communication-oriented personal computer environment. (We also have the option of going in what we think are the best directions technically and hoping we can drag the rest of the world along with us.)

In the case of Mesa, the conflicting factors are our desire to benefit from the large amount of work that SD will be doing on the language and system, and the differences between our needs and theirs.

LRG's plans for Smalltalk agree with our desires in many areas, but again their priorities are different. In particular, they are much more interested in small systems and in computer-naive end users.

PROBABLE EVOLUTION

Lisp

Danny Bobrow prepared a discussion of directions in which the Lisp environment will probably evolve if there is no special PE effort (on [Ivy] < Deutsch > per > future > epe-features.memo). At least some attention will be paid to several of our high-priority items, so things aren't quite as orthogonal as with Smalltalk, but neither will it be anything close to what we have been hoping for and discussing. The following is a condensation of the major points from Danny's memo.

(L6, L23) (processes, monitors, interrupts) Assuming that Pilot is easily available, Pilot's features in this area will be supported, including the ability to resume a process independent of what language it is written in.

(L8) (packed data) Facilities for user data types, including packed data, will be made available. There will be nothing to compare with Mesa's use of packed data; Danny doesn't believe this is important in building experimental programs on the Dorado.

(L11) (static type checking) Larry Masinter's thesis is likely to extend DeclTran to inter-function consistency checking. Mesa-style optimization is unlikely.

(L13, L25) (encapsulation) Something like the block compiler will get into the Dorado system, perhaps along the lines Danny and Peter worked out for the proposed XL system.

(L14) (abstraction) The type checking provided as a result of Larry's thesis will make it possible to provide interface functions that resemble the current Mesa idea of an interface as a set of type specifications.

(L21) (integrated sublanguages) This is an area in which Lisp users have strong feelings. Here are Danny's comments in full:

> The ability to create sublanguages is one of the most important facets of Lisp programming as it is done. Interlisp itself has almost a dozen such sublanguages. KRL, ATN's, etc. are all examples of how we capture our basic ideas in an abstract language which we want to use in conjunction with Lisp itself. I would rate this as an A priority item. Currently Lisp has the best facilities for building such languages, but full integration is a way off. In the next two years I see much more work done on this issue, making it even easier to drop into a dialect, and through that back into Lisp while passing the appropriate contexts.

(L30) (control and data trapping) KRL will be extended to make it cleaner and more efficient. There is some chance that more primitive facilities will be pushed down into the record package.

Mesa

Ed Satterthwaite contributed a long memo (on [Ivy] < Deutsch > per > future > mesaprospects.bravo) discussing probable future SDD activities towards improving the Mesa environment. The following is a condensation of his overall remarks.

> In the short term, any incompatible changes to the virtual machine or language will be strongly resisted unless the benefits of such changes are very substantial and very obvious. In the longer term, SDD should be interested in helping to design and implement significant extensions in these areas when they address acknowledged problems of interest to SDD (development of the process stuff provides a good model of how this might happen).

A lot will be done to improve the performance of all components of the Mesa system. I believe that the compiler, for example, will respond well to tuning and cleanup, especially on the Dstar.

For obvious reasons, there is interest in more extensive optimization (both speed and space), as in item (T13), and in static analysis that is likely to reveal lurking bugs (e.g., range analysis, uninitialized variable detection), as in item (T19). I (Ed S.) am personally interested in doing these sorts of things.

A lot of effort will go into tools. My feeling in the past has been that SDD's tools are likely to be somewhat heavy-handed in providing administrative control; I have come to appreciate the controls for large, multi-person projects but am not convinced I would like them for a one or two person experiment.

SDD-produced products will clearly provide many of the functions we would like in packages, but it is unrealistic to expect packages from the product program except by luck (it might be possible to extract and repackage chunks of code). Collaborative development of packages with the prototype software group (who also need an (EP)E) seems more likely and promising.

Ed noted the following items as likely to receive SDD attention in the relatively near future:

(L8) (packed data) Better record packing, bit arrays.

(T2) (cheap interpreter) A packaged version of the interpreter currently present in the debugger.

(L12) (self-typing data) Sequences, cleaner variant records, some help with initialization and finalization; some language support for object-oriented programming, which is already in wide use as a programming style.

(L25) (importation control) A known significant source of obscure bugs.

(L24) (simple syntax) (T4) (prettyprinter) The latter should help with the former.

(T1) (fast turnaround) Considered essential; being able to replace a module in a running system is already planned.

(T5) (consistent compilation) (T6) (version control) Considered essential; being studied by a high-powered working group.

(T9) (dynamic measurement) Primitives exist, packages are needed.

Smalltalk

Discussions with members of LRG suggest that they are most likely to work in the following areas:

Development of a constraints interface, following Alan Borning's work, possibly including pushing some of this down into the lower implementation levels.

Multiple superclasses.

Simplification of the user interface, following Adele Goldberg's ideas.

A major effort to fit the system into small machines, specifically Notetaker.

Information retrieval capabilities.

More work on media, such as music and animation.

Some work on parallel execution by multi-processor machines.

Aside from the information retrieval area, which might address item (L3), this work does not relate directly to our high-priority items. Of course, the Smalltalk environment will also continue to improve in unanticipated ways as LRG builds tools for themselves.

<Deutsch>PE2>memos.press

Inter-Office Memorandum

To	EPE Group	Date	September 12, 1978
From	Daniel G. Bobrow	Location	Palo Alto
Subject	Some Lisp features I would be loath to do without; or would you take a fish without a bicycle	Organization	CSL

XEROX

Filed on: <Bobrow>lispfeatures.memo

I have been introspecting on how we have used Interlisp to implement and embed KRL, and the following are my distillations of the features used which I would want to have in our EPIC (Experimental Programming Integrated Colossus). To make these next points more specific, here are some fragments from a particular version of KRL-1.

Here is a KRL unit taken from a cryptarithmetic problem as it appears on a file.

```
# Column
  self:↑1                    1:Trigger(ToPrint,'(For x in '(top bottom sum)
                                        do (PRINT (SeekMy 'Pointer SomeSignalTable))))
    top: An integer
    bottom: An Integer
  sum:↑2 An Integer                2: Trigger(ToFind, '(LispForSum INSTANCE))
```

Here is a fragment from the middle of some Lisp code that might be run using some mixed KRL and Lisp syntax.

```
(SETQ SUM
      (Seek 'Pointer
            \the summ fromA Column thatIs !Post (CAR columnList)//
            FullMatch))
```

1. Variable Syntax Analysis: That is, procedural escapes in an available parser. The built in Lisp parser (a structure parser that knows about parentheses, brackets, and lexical analysis) also allows access to the basic reading routines (read a character or a symbol or a ...) and allows certain characters to signify procedural escapes.

In the example above there are two different forms of variable syntactic analysis illustrated. In the unit there are examples of the embedding of Lisp expressions; thus a special parser for units must be able to call on the Lisp standard reader. Secondly, in the middle of the code fragment is a "\" indicating an escape to the KRL reader.

2. Programmable Lexical Analysis: Another name for being able to read in atoms under some control (I don't think Lisp's readtables are optimum). Reading results in a unique pointer associated with that string. The unique pointer must be able to be used in various kinds of symbol tables and programs eg

1) To access a procedure definition which can then be run; in the example LispForSum is a procedure to be run.

2) To obtain a value (current under some scoping); In the unit, the variable INSTANCE is assumed bound in the interpretive context.

3) To be compared to a constant in a program; eg the first argument to Seek is a constant which is going to choose a case.

3. Procedural Embedding: This consists of two parts; the use of program text in the context of other structure (the problem of getting in and constructing structures which can be treated as expressions to be evaluated); and the evaluation of said structures in a specified envirionment. In a compile time system, I believe it would be possible to fix the environment as one specific to the interpreter, and/or pass to the interpreted code a data structure which captures the appropriate state. The program structure used should be analyzable and evaluable. Two different kinds of embedding are illustrated: in the unit a prgram fragment in real Lisp as used by the Triggers; and in the code fragment, the ability in Lisp to build a special purpose compiler which can substitute for the Seek expression containing a constant data structure argument some other arbitrary Lisp expression which is then compiled or interpreted by the Lisp system.

4. File Pointers: This is my key phrase for being able to refer to a large extended address space from within Lisp. This is used in KRL for example to point back to the original symbolic version of a unit on a file from the in core version, and the update package to copy symbolics when making incremental changes to a file.

5. Hash Linking: Optional fields in data structures are used often in KRL, and are implemented by using the address of the data structure as a key in a hash structured symbol table. For example, pointers to the triggers for a slot are stored in this way since most slots don't have triggers.

6. Spelling correction: Knowledge by the system of names appropriate for certain environments, and correction of the name I typed to a close one which is appropriate. This is especially useful in correcting names of data fields in units.

7. Evaluation in editing context: To test changes in an edited function, I sometimes select a particular form which I have isolated as having a bug, change it, and want to evaluate it in the context of the program as reflected in the program stack at the time the bug manifested itself.

8. Simple linked data structure. Lists are a super first class data type in Lisp, and their existence should not be ignored, nor the wide range of library functions to manipulate same.

9. Both for lists and other data types we depend on the system maintaining storage for us, providing automatic allocation and deallocation.

Here are my interpetations of some of the comments I got from Doug Lenat about what he likes about Interlisp:

10. The thickness of the manual: There are few indispensable functions, but throwing away any 25% would be a blow to my programming style.

11. Portability of knowledge: AI programmers come knowing some brand of Lisp, and immigration into Lisp is therefore much easier than into some other language.

12. Use of internal structure of names. I explode atom names so that I can search for patterns in them. I also generate plausible names to go with new concepts defined, and programs constructed by my program.

13. Programmable Interrupts: I use the abaility of Lisp to make single characters force a call on a function to monitor and control the the computation in my system.

14. Personalized macros: I added some user macros to personally tailor the editor to my style.

15. Interaction monitoring: I used the easy facility to have typescript files to keep hardcopy records of my interactions.

16. Performance analysis functions: I used **breakdown** (which rewrote my functions to monitor their performance) to tune and understand my system.

17. Programmable debugger: Conditional monitoring of function calls and returns, and firewall protection (ERSETQ) was important in getting my large system up.

Inter-Office Memorandum

To	EPE Working Group	Date	September 14, 1978
From	P. Deutsch (including comments by McDaniel and Swinehart)	Location	Palo Alto
Subject	Unique Smalltalk capabilities	Organization	CSL

XEROX

Filed on: <Deutsch>stcap.memo on Maxcl

This memo is a first stab at listing in detail those capabilities of Smalltalk which are not present in either Lisp or Mesa, and which are important for an EPE. Comments by Gene McDaniel and Dan Swinehart have been incorporated in the memo without attribution.

Language

>>Syntax: Smalltalk has an infix syntax describable in less than 2 pages of equations, including the lexical syntax. It has no reserved words and only about a dozen reserved characters. Smalltalk's syntax provides the following advantages:

(1) The syntax can be learned very quickly by anyone who knows any formal language, and there is never any need to refer to the manual to decide whether a given syntactic construct is legal.

(2) It is possible to design a very simple internal representation of parsed programs, which can be effectively manipulated by programs.

(3) The syntax includes the capability for certain kinds of extension such as user-defined control structures.

(4) There is a uniform notion of "application" which encompasses all language constructs that carry out operations: this means that a future macro-expansion capability will be very easy to add.

(5) The use of punctuation for the common operations (statement grouping and conditionals) enhances compactness without sacrificing readability.

One consequence of this simple syntax, of course, is that a lot of things which one thinks of as being part of the "language" in Mesa become "packages" in Smalltalk (just as in Lisp), but that is just a matter of how the documentation is arranged.

Can the Mesa syntax be simplified to include the above advantages? Can the Lisp infix syntax provide (1)? Can the tangle of Lisp compile-time capabilities (macros, CLISP expansion, etc.) be simplified to provide (4) without giving up (3)?

>>Object-oriented programming: in Smalltalk, all objects interpret the operations legal on them. There is no notion of a free-standing "procedure" or "data structure". This has the following advantages:

(1) The space of operation names is local to each type -- there is no need to qualify names (as in Mesa), or invent names that include the object type in them somehow (as in Lisp).

(2) There is no need to declare the type of the receiver of a message, or to qualify or "open" the names of its components as in Mesa.

(3) All operations are automatically generic, with efficient support from the underlying virtual machine.

A consequence of (1) is that Smalltalk syntax may be a reasonable model for certain application languages as well: in particular, there is no need to provide a command interpreter that looks up a special command set in a table, and eventually spelling correction and type checking could be done by the language mechanisms as well.

Can Lisp be modified to include appropriate mechanisms for name scoping and interpretation for (1) and (2)? Can either Lisp or Mesa provide (3)?

>>Hand coding: Smalltalk provides no way for programmers to write code in any language lower than Smalltalk, and could easily do without the loophole which allows programmers to access arbitrary memory words directly. In Lisp, by contrast, programmers constantly give in to the temptation to write assembly code and/or direct data accesses for efficiency, which result in obscure bugs and machine dependencies.

Can Lisp be modified to provide a sufficiently complete set of non-machine-level data access mechanisms to make assembly code unnecessary? Can Mesa be modified to remove or greatly constrict the loophole mechanisms (including dangling pointers) that compromise program correctness in obscure ways?

Can either Lisp or Smalltalk be upgraded to provide a compiler good enough to reduce the temptation to hand-code to an infrequent, rather than a daily, level?

>>Inheritance of operations through subclass mechanism: Smalltalk makes it relatively easy to create classes of objects which are specializations of more general classes (e.g. Atom specializing String specializing Array), or which supply the implementation for a generically defined class (e.g. Integer and Float implementing Number). Work is going on now to extend this to more complex ways of combining behaviors into new ones. This inheritance idea includes at least some of the Mitchell-Wegbreit "schemes" ideas, as a way to specify shared behaviors only once, and seems to reduce the amount of code one needs to write to make use of existing classes.

Lisp has no class structure at all; can Mesa's notions of interfaces and modules be extended to provide the flexibility of current and promised Smalltalk subclass mechanisms?

>>No second-class citizens: Smalltalk makes absolutely no distinction between system- and user-defined types or operations, except that certain operations on system-defined types are provided in lower implementation layers.

Can Lisp be extended to provide the benefits of allocation, access, printing, checking, and special compilation of user-defined types and their operations that now are the exclusive province of the built-in types?

>>Future directions: some experiments in future expansion of the notion of programming language are being carried out in Smalltalk now. How hard would it be to do these experiments in Lisp or Mesa?

The proposed template style of two-dimensional object description.

Constraints (Alan Borning's work). Dan Swinehart has been doing some display interfaces (e.g., audio board test program) using Keith Knox's package as a base, where user changes in one entry must cause changes in others -- and vice versa. When combined with the desire to initialize these entries with default values and a few other things, he found the resulting ad hoc programs to be complex and cumbersome. A constraint language would be ideal for specifying these relationships.

The proposals to embed data base linkages as primitive notions in textual objects, such as source programs. (This might be a way to handle some of the linkages needed by Masterscope.)

Tools

>>Browser: all system and user source code is instantly available for browsing, including documentation and table of contents per class, and class- and method-level categories.

Lisp goes part-way towards this, in that it is relatively easy to view the source code of one's own programs. Can Lisp or Mesa make the system code equally accessible? Can Lisp or Mesa incorporate (and require use of) the category system, which greatly helps finding one's way around in one's own and others' programs?

>>Integrated on-line documentation: the programmer can easily insert documentation in the code as it is written or edited. (Off-line documentation is admittedly awful -- it could be improved even by some simple automatic extraction of on-line stuff.)

>>Immediate incremental compilation.

Can the Mesa compiler be made "instantaneous"? (Smalltalk compilations, even on the Alto, are not a major computational bottleneck.) Can the Lisp compilation process be made truly invisible and automatic without giving up the fast turn-around from editing to execution?

Packages

Arbitrary-precision integers with smooth transitions.

Paragraph formatting for display and Press output.

Bitmap and display coordinate manipulation.

Paragraph and picture (bitmap) editor.

Windows with standard behavior -- used by interactive tools.

Menus.

Other

Non-preemptive window behavior (consistent model of interaction). Makes mutually non-interfering applications easy.

28-SEP-78 13:53:02-PDT,17821;000000000000
Date: 28 Sep 1978 1:48 pm (Thursday)
From: Horning
Subject: Re: A reminder
To: Deutsch

Subject: DRAFT Mesa -> Lisp Requirements
To: Programming Environment Interest: Bobrow, Deutsch, Horning, Lampson, Levin, McDaniel, Masinter, Mitchell, Morris, Satterthwaite, Suzuki, Swinehart, Taft, Teitelman;

DRAFT - DRAFT - DRAFT - DRAFT - DRAFT - DRAFT - DRAFT - DRAFT

[This is a draft, being circulated now for comment. Sept 28, 1978]

[We recognize the considerable merits of the Interlisp programming environment. This document is intended to point out what we consider to be the correspondingly major advantages of the Mesa environment. We feel strongly that these features should be preserved in any EPE. To sharpen the discussion, we have made our remarks quite pointed. We trust that in most cases our "challenges" can be met (with varying difficulty); the reason for posing them is to ensure that they are not overlooked.]

[Items drafted by Horning:]

>>Static Checking

ISSUES: One of the major advantages of the Pascal family of languages (of which Mesa is a member) is the substantial amount of consistency checking that is possible statically (i.e., independent of program execution). Examples of this are syntax checking, which catches many transcription/input errors, and checking for declaration of all names used in a program, which catches both spelling errors and misunderstandings of the environment. However, probably the most important form of static checking is for type consistency, which is very effective at catching many forms of errors, including many "logical" errors.

There are several reasons for preferring static to dynamic checking:
- -it occurs sooner, thereby giving quicker feedback;
- -errors are more easily related to their sources;
- -many errors can be detected in a single run, rather than being identified one at a time;
- -complete checking is not dependent on executing an exhaustive set of test cases (once a program is accepted, one KNOWS that it has no lurking bugs of this sort);
- -the overhead of checking can be paid once, rather than repeatedly at execution time.

If programs developed in the EPE are to achieve laboratory-wide usage (like Bravo and Laurel), then their robustness cannot depend on run-time interaction with their developers. (Not only is there no debugger in Peoria, there generally isn't one here, either!)

Many Mesa programmers are willing to accept "long" turnaround times for program changes. Perhaps one reason is that they have much more confidence that their programs will actually work once they are accepted by the Mesa compiler.

CHALLENGE: How can the LISP system be modified to allow a comparable

amount of static checking? In particular, can static tests be performed that ensure freedom from type errors and use of undeclared names?

>>Static Scoping

ISSUES: Mesa and LISP have quite different rules for accessing non-local ("global") variables. In Mesa, the association between name and variable (or, if you prefer, access algorithm) is determinable statically, and is a function of the procedure declaration and its environment. If this association cannot be fixed, it is difficult to develop robust procedures to manipulate data structures that out-live particular procedure activations. It is completely unacceptable for accidental re-use of some name in a calling environment to affect a procedure's access to its non-local data.

Scope rules serve another very useful function. They provide two sorts of limits:
- -they limit the set of names accessible within the program at any given point (the amount of "relevant environment" that the programmer need consider), and
- -they limit the range of text within which a name can be accessed, and hence the amount of text that need be considered when considering a change or searching for a bug involving the named entity.

CHALLENGE: How can LISP provide a satisfactory mechanism for statically associating names and non-local variables? How can names be suitably localized?

>>Encapsulation

ISSUES: Even packages that have a large amount of local state should have narrow interfaces. Ideally, the user should know (and use) only the published interface.

In Mesa it is possible to declare items (data, functions, procedures, types, etc.) to be PRIVATE to a module, and inaccessible outside. This is invaluable in building packages, because it ensures that only the PUBLIC interface is used by clients; assumptions made by the implementation cannot be violated from the outside through either inadvertance or malice.

Mesa allows a program to create multiple instances of a module, supplying, if desired, different parameters to each instantiation. This supports an "object oriented" style of programming, in which each object is referenced only through its associated suite of functions and procedures.

CHALLENGE: How can LISP provide an adequate encapsulation facility for packages? Can LISP support multiple instances of package prototypes, and object-oriented programming?

>>Explicit Interfaces

ISSUES: Mesa provides a special type of module (a definitions module) specifically for the purpose of declaring interfaces. This makes it possible to separate the development of implementing and client modules; with few exceptions, the changes that don't affect the definitions module are invisible to the client, and those that do require corresponding changes in the client.

Explicit documentation of interfaces, of course, is good programming style, and mandatory for effective multi-programmer cooperation. Static checking that

interfaces are adhered to is a bonus.

By separating the definitions module from the implementing module(s), Mesa makes it convenient to test a module in an environment designed for the purpose, and have some assurance that it will continue to work properly when substituted into a larger system with the same interface.

CHALLENGE: How can explicit, enforced interfaces be introduced into LISP?

>>Consistent Version Checking

ISSUE: Programs are developed through many versions. The problems that result from inconsistent versions can be extremely subtle, yet it is not always easy to ensure that a set of components being run together are actually consistent. This program is alleviated (but not completely solved) in Mesa by enforcement of the rule that the components must use the same versions of interfaces (definitions modules).

CHALLENGE: How can LISP prevent the accidental execution of systems built from inconsistent versions of components?

>>Efficiency

ISSUE: One of the key principles of Mesa is that the programmer shall have effective access to the full power of the machine, and that implicit run-time overheads imposed on all programs shall be kept to a minimum. It seems clear that there will always be some experimental programs whose computational requirements will equal (or exceed) the power of the available hardware, and hence that it will always be necessary to avoid implicit overheads.

One advantage of a (reasonably) optimizing compiler is that increased efficiency can be obtained with little or no effect on the program form and semantics by supplying tighter (or earlier) bindings (e.g., by replacing variables by constants).

CHALLENGE: Is there a LISP subset within which the full power of the machine is available and implicit overheads such as garbage collection and indirection on data reference are eliminated? Can the efficiency of some LISP programs be made comparable to Mesa by simply accepting earlier bindings?

>>Exportation of Programs from Environment

ISSUE: Programs developed in the EPE must not be restricted to use within the EPE. Mesa provides a mechanism (image files) for producing essentially standalone systems that carry along with them only those parts of the Mesa world on which they rely. This is important because users may not be Mesa programmers and may not be willing to pay the overhead of keeping the Mesa world in their machine. (E.g., the typical Laurel user should neither know nor care that Laurel is written in Mesa.) Among other things, this means that it must be possible for the program itself to intercept all error reports, and dispose of them appropriately.

CHALLENGE: Can LISP programs be packaged for standalone use? Is it possible to use systems written in LISP without either knowing anything about LISP or paying the overhead of the LISP system? Can LISP programs be made to process all internal and system error reports?

[Items drafted by Mitchell:]

>>Exception Handling

ISSUE: There are three situations in programs that correspond to exceptions:
(a) impossible states from which recovery is unlikely (usually an indication of a bug),
(b) low-frequency states from which the program should be able to extricate itself (e.g., difficulty in reading a page from a disk), and
(c) in interactive systems, user-initiated exceptions (e.g., kill this procedure call and the two previous and then retry).

In case (a), whatever mechanism is used to inform the programmer (assuming he is there) of the error must have the feature that it does not destroy any of the state information that he needs to investigate the bug. In particular this means that the execution stack should not automatically be rolled back as part of raising the error.

In case (b), the errors that a procedure may raise should be part of its interface specifications (they are not currently in Mesa). The program that is selected to handle the exception must be able to receive information (parameters) along with the exception notification, and it must be able to permanently acquire control in order to terminate the action. This will require rolling back the execution state (note that I am very carefully avoiding the phrase "cutting back the stack" - that is not what is needed ina multi-process environment). Such execution unwinding must be done in such a way that each outstanding activation that is to be terminated have a chance to handle the fact that it is about to die; this makes the decisions about what special actions are needed for termination a local one within the activation being killed. The order of cutback must also be reasonable; e.g., in a procedural environment, they must be done in a LIFO order, sort of like backward execution.

Case (c) raises no issues that (a) and (b) have not already proboked. It is included here only for completeness and because an integrated EPE will need to do this.

CHALLENGE: Can Lisp provide an error-handling mechanism with local handling of termination and with the state-preserving property for debugging purposes?

>>Processes and Monitors

ISSUES: Processes and monitors provide a well-engineered way for introducing parallelism in one's programs. I doubt that they are the ultimately best way for doing so, but they represent the state of the art and there is much experience with using them and with proving properties of programs that use them.

CHALLENGE: What does Lisp have?

>>Coroutines

ISSUES: Some form of coroutines are needed to handle situations such as text formatting. The Mesa control structures are built from a common underlying primitive that allows one to call another program in a uniform manner without being aware of its control regime. The called program could be implemented as a procedure or as a coroutine (there is more than one flavor). This makes a control abstraction facility that is useful for object-oriented programming, since the user of an object can treat all operations as if they are procedures no matter what their exact implementation is.

>>Records, Parameters, Constructors, and Extractors

ISSUES: Record constructors are most frequently used as parameter records to procedures in Mesa. Keyword notation eliminates errors of ordering of two or more parameters of the same type (e.g., COPY[from: x, to: y, nWords: 256], for which I could never remember the correct order of "to" and "from" in the positional notation). Making records first-class values requires that one be able to write record-valued expressions. Programs are clearer when an entire value is specified at once, rather than by a number of assignment statements. Also, writing a constructor enables the compiler to check that one has specified some value (possibly "don't care") for each component and enables one to have default values for some componenets. This also provides a way of having default parameters for procedures as well.

Procedures need to be able to return multi-part values and not be limited to a single value returned in order to avoid some uses of "output parameters." Mesa extractors then allow the programmer to perform multiple assignments of those multi-part values to separate variables for further processing. This is heavily used in Juniper.

CHALLENGE: Can the Mesa record/parameter facilities be provided in Lisp?

>>Efficient Random Access to Files

>>Debugger Access to Source

ISSUES: One of Mesa's strong points as a system programming language is that the programmer can debug at the source level. This requires the ability to specify and set breakpoints by reference to the source program (either by context or by pointing). If the Mesa compiler were to do more global optimization (such as Fortran H's, for instance), the programmer would still need to be able to compile his program in a mode permitting the correspondence between source and object probrams to be maintained.

CHALLENGE: I presume that Lisp has these features normally. What about the block compiler?

[Items drafted by Suzuki:]

>>Readable Programs

ISSUE: It is important that one can express his algorithms clearly so they can be easily understood by others. This is a non-trivial aspect of language design. Mesa has some features that help people to write clear and understandable programs.

Mesa provides the generally accepted control and data structures in forms whose syntax corresponds well with their semantics. It follows many accepted conventions of mathematics and conventional programming languages. Infix operators, precedence of operators, and structural keywords all eliminate the need for many parentheses and reduce the parenthesis matching problems faced by LISP users.

CHALLENGE: Can LISP get rid of the parentheses and enhance readability of programs without falling into ambiguity?

>>Precise Syntax

ISSUE: One advantage of Mesa is that the syntax is rigorously defined so that when one writes a program it is known a priori whether it is acceptable or not and how it is parsed and executed.

In CLISP the syntax is tied to the error checking mechanism in order to implement language extension. Therefore, it is not easy to know in advance whether what one writes is acceptable and even accepted whether it will be parsed as the programer wishes. If the program is not accepted or misinterpreted, then it may be difficult to guess the style of writing which CLISP likes.

CHALLENGE: Can LISP be liberated from error-driven syntax? Can a precise, yet reasonably compact, syntax for acceptable programs be given?

>>Enumerated Types

ISSUE: A very useful tool in programming is the ability to create a type whose elements have programmer-defined names. One example is a parse tree in a compiler where each node belongs to one of a finite number of classes like "assignment" and "addition".

Mesa enumerated types provide this facility. They offer the following advantages:

-efficient use of storage,
-usable as array indices and for iteration control,
-safe when used as array indices (no subrange overflow),
-compact notation for subsets,
-combined with Case selection, we can attain an efficient implementation of multiple-branch control mechanism.

CHALLENGE: Provide an equivalent facility in LISP.

>>Case Selection

ISSUES: As noted above, Case selection in conjunction with enumerated and subrange types can lead to efficient multiple branches utilizing the hardware dispatch control mechanism.

Another feature of Mesa SELECT statements is that one can write relation tails as selection criteria so that evaluation of the first operands can be saved.

CHALLENGE: Provide an equivalent selection capability in LISP.

>>Control of Storage Structures

ISSUES: Mesa makes it possible to increase storage utilization (possibly at the expense of access time) by using the PACKED option. Fields can be as small as one bit.

Mesa also enables the precise control of the layout of fields within a record for dealing with externally-defined structures (e.g., interrupt words).

CHALLENGE: Can LISP provide equivalent density and control where required?

>>Constant declaration

ISSUE: Constant declaration is a mechanism to bind a name with a value throughout the execution of a program, so that no statements can overwrite the

value. The mechanism is very useful in constructing reliable software, since this property can be checked locally.

There are two levels of constant declarations in Mesa. One is "module constant" by which a name is bound to a unique constant throughout the life of a module. One cannot even overwrite the value. The other is "local constant" by which a name is bound to a value, which may be different for each procedure invocation, but cannot be overwritten.

CHALLENGE: How can LISP ensure the security of name-constant bindings?

DRAFT - DRAFT - DRAFT - DRAFT - DRAFT - DRAFT - DRAFT - DRAFT

Inter-Office Memorandum

To	EPE Working Group	Date	September 21, 1978
From	P. Deutsch, D. Bobrow	Location	Palo Alto
Subject	Mesa -> Lisp reply	Organization	CSL

XEROX

Filed on: <Deutsch>lispreply.memo on Maxc1

This draft responds to the challenges offered in Jim Horning's memo "DRAFT Mesa -> Lisp Requirements" of 9/14/78 (including Jim Mitchell's addendum) and Peter Deutsch's memo "Unique Smalltalk capabilities" of the same date.

Premises

There are certain modifications and additions which we agree must be made to Interlisp (for the Dorado environment), and which impact several of the items in the lists below, so we will enumerate them in advance.

We agree that substantial work needs to be done on the compiler. The byte compiler is much cleaner and simpler than the PDP-10 compiler, and we believe that the additions to handle type declarations (including variant representations) and name scopes are straightforward.

We agree that the notions of interface and scope need to be made explicit in the Lisp system in the form of objects with well-defined behavior -- used by the compiler, Masterscope, the file facilities, etc. Larry Masinter believes that adding a Mesa-like notion of interface to Masterscope is straightforward. In particular, we believe that the Mesa arrangement of separating interface declarations from both client and implementor should be supported. Our current idea for supporting both these notions and fluid variables (see next paragraph) is to introduce an object called a dictionary, which associates interpretations with names in a given scope. (We have a number of design memos that expand on this idea.)

We believe that it is appropriate to replace the current Lisp notion of free variables with a combination of lexical scoping and declarations for fluid variables (variables dynamically bound by name as at present). The MIT SCHEME dialect of Lisp has done this successfully, and an interesting and efficient compiler for this dialect exists. Our present implementation can support this change, but at great cost in runtime efficiency. SCHEME shows one way to improve the efficiency at the cost of a fairly radical implementation change; we do not yet know exactly how we will want to change our implementation.

Both for lexical scoping and for other reasons, we believe we need to provide an efficient mechanism for FUNARGs, or, more generally, for <code, partial environment> objects which can be applied as functions. Such objects are generalizations of current Mesa procedure objects.

We agree it is necessary to support a precisely defined infix syntax in a clean way. Our intent is to provide some general-purpose parsing and unparsing (prettyprinting) mechanisms to handle the syntax, and to define all our semantics in terms of extensions to S-expressions. Depending on how much non-local declarative information is needed for the translation, we may defer translation until compilation, until the user requests it, or until the first attempt to execute the function, but in any case the entire function will be translated at once in a precisely defined way that does not depend on

the dynamic environment.

Horning's items

>>Static checking (for type errors and undeclared names)

Lisp already has a type calculus approximately as rich as that of Mesa (the type system done by Kaplan and Sheil for DECLTRAN), including an S-expression syntax for declaring argument, local variable, and result types. Currently checking is only done within a single function. Adding cross-function checking to Masterscope should be straightforward logically.

Masterscope already has the capability to check for undefined names, both undefined functions and variables which might not be bound.

>>Static scoping

See "Premises" above.

>>Encapsulation

There are some hard issues here -- Lisp doesn't lend itself easily to the kind of total encapsulation that Mesa does. To the extent that lexical scoping conceals internal names, we can do a reasonable job of keeping clients out of implementations. Linked function calls are an existing mechanism for removing external access to local functions.

Multiple instances of modules can be accomplished using FUNARGs. See "Premises" above.

For object-oriented programming, see the discussion below under the Smalltalk list.

>>Explicit interfaces

See "Premises" above.

>>Consistent version checking

We can use the same time stamp scheme that Mesa does.

>>Efficiency

Given the existence of the type system, including the (already existing) provisions for packed records, the compiler can potentially do as well as Mesa for manipulating scalar data. (There are some issues here about instruction set optimization, however.) We believe that a combination of compiler analysis and user declarations can make possible the Mesa-style stack allocation of composite objects with direct addressing, without running the risk of dangling references; we recognize this is not a trivial problem, however.

>>Exportation of programs from environment

We believe that this issue is partly a red herring: after all, even a Mesa.Image file contains copies of all the system modules needed by the program. The irreducible minimum amount of "system" for a Lisp program is about the same as that for Mesa, except for the garbage collector. Given that Masterscope can take a fully bound Lisp program and find all the functions that are callable in any way, we can certainly produce a virtual memory image file that contains only the necessary functions (in fact, we should be able to do better than Mesa since we don't need to take entire "modules").

System overhead during execution is essentially limited to garbage collection and other storage allocation functions. The arguments under Efficiency above suggest that it is possible to incur this overhead relatively rarely, and of course if a program does no dynamic allocation, even the garbage collector (a Lisp program) need not be included in the image file.

Lisp programs can currently intercept all software errors including dynamic type errors, array bounds errors, arithmetic overflow, etc. Lisp programs are not currently able to intercept and recover from hardware errors or running out of storage. Mechanisms to preallocate storage and to pin code and data into real memory, which seem necessary to take care of these, could be added to Lisp with no fundamental change.

Suzuki's items

All of these items are essentially syntactic. All are either already present or very easy to add.

>>Readable programs; precise syntax

See "Premises" above.

>>Enumerated types

The DECLTRAN type system already has this at a semantic level; extending it to the representation level should be easy.

>>Case selection

This is a modest generalization of the present SELECTQ function.

>>Control of storage structures

Lisp already allows tightly packed records. Adding explicit user control over the layout is trivial.

>>Constant declaration

The compiler already provides for compile-time constants. Local constants (not changeable after initial binding) are trivial to add.

Mitchell's items

>>Exception handling

Lisp already funnels all exceptional conditions through a single point (the ERRORX function). These conditions look like a normal function call, so no state is lost. ERRORX can easily be changed to examine the execution environment and call cleanup handlers associated with intermediate frames, in any desired environment (the dynamic environment of the error and/or the environment of the intermediate frame).

>>Processes and monitors

Lisp currently has no process or monitor machinery. Processes are easy to implement with the same mechanism as coroutines (see below). Monitors can be done the same way as in Mesa, provided that object-oriented programming is enforced (see the discussion under Smalltalk below).

>>Coroutines

Lisp currently has a notion of an "environment descriptor" which is essentially an indirect pointer to a stack frame. Functions exist which provide coroutines using a single environment descriptor that gets switched between the participants. A more general coroutine mechanism, that allowed coroutines to look like function calls and returns, could easily be built, and made efficient with a little microcode assistance (similar to the microcode assistance we will want for FUNARGs).

>>Records, parameters, constructors, and extractors

Records already exist. Extractors are a pure syntactic extension. Keyword notation already exists for constructors; adding it for procedure parameters is fairly straightforward, even if no declaration exists for the callee (the names must be matched up at link or call time in that case). Providing multiple return parameters (as opposed to a heap-allocated return record) requires some tinkering with the instruction set.

>>Efficient random access to files

There is no reason why Lisp and Mesa should be any different in this respect -- in fact, it is planned to provide microcode support in Lisp for I/O streams.

>>Debugger access to source

Lisp already provides for access to the source program. There is no current provision for putting a breakpoint in the middle of compiled code: internal breakpoints cause the function to revert to interpreted form. Converting a PC back to a location within the source can be difficult, because Lisp functions often have no statement structure in the Mesa sense (when they do, we can easily produce something like the Mesa fine-grain table): for a non-optimizing compiler we could do it by repeating the compilation and stopping when the PC matched.

Currently, the debugger cannot show frames (either function names or variable values) within a compiled block. It is logically possible to retrieve this information, although about as messy as for the Mesa debugger.

Smalltalk items

>>Simple, extensible, macro-expandable, infix syntax

See "Premises" for syntax. Once we get rid of CLISP and introduce a single uniform way of dealing with non-standard S-expression syntax (like SELECTQ, PRINTOUT, etc.), we will be doing as well as Smalltalk. We believe it is extremely important to do away with the current notion of compiler macros, replacing it by a scheme in which one gets the choice between closed calls, open calls (using the *same* definition), and macro-expansion (an arbitrary user-supplied function, the same for interpretation and compilation, computes the form to be executed).

>>Object-oriented programming (operation names local to each type, all operations generic, no need to declare types, instance state automatically available)

Generic operations, decoded according to the receiver type, can certainly be provided in Lisp now, at considerable execution cost. The real issue is how efficient various degrees of binding must be. The current Lisp approach is to introduce one level of indirection for function names in the normal case, removable by declaration (linked calls), and no discrimination on the receiver type. If an operation is declared generic, we can certainly provide microcode assistance for getting to the right piece of code.

Making the instance state automatically available comes for free with an efficient FUNARG mechanism. However, we believe that the right approach is to go down the Mesa path of trying to eliminate the distinction between variable binding records and other kinds of records. We do not yet know what this means in terms of implementation structures; at the S-expression level, it

certainly means the ability to "open" records for direct addressing exactly as in Mesa. This "open" for instance variables can be inserted automatically when one uses an object-oriented interface.

The scope restrictions associated with object-oriented programming have already been covered under Horning's comments. Private record declarations are the key to this. We do not know exactly how we will make it hard enough for ordinary clients to penetrate an implementation, while making it easy enough to examine (and modify?) state with the debugger.

A more serious question is whether there are benefits from the object-oriented style that one gains only if nearly all programming is done that way. We do not propose to rewrite the Lisp system in object-oriented style, although we think that most new packages will be written this way, and some incremental rewriting will probably occur.

>>Inheritance of behavior through subclass mechanism

We think this can be provided through the dictionary mechanism mentioned under "Premises".

>>No distinction between user- and system-defined types

To the extent that programs are written in object-oriented style, there is no reason why the mechanism for handling operations on integers, say, should be different from that for user-defined types. There is, however, a serious question as to what the default behavior of the compiler should be: should a call to an undeclared function (i.e. a function for which there is no corresponding interface declaration) be treated as generic or not? Addition is presumably an example of such a function, if the argument types are undeclared. Making addition work properly for integers, long integers, floating point numbers, rationals, algebraic expressions, etc. requires generic treatment. We do not have a good answer for this.

>>Future directions: template-style programming, constraints, data base links in text objects

Given Lisp's hospitality to programs that construct and manipulate programs, template-style or constraint-oriented programming should be even easier to implement than in Smalltalk. Data base links in text objects (such as source code) require careful planning of the interface to the text manipulation system, but again Lisp has a better package for this purpose (TXDT) than either Mesa or Smalltalk.

Tools

>>Browser

The Smalltalk Browser depends on having all the system sources and documentation available on-line, and on the 3-tiered classification system (class category, class, method category). Class categories correspond essentially to chapters in the Lisp manual. Classes correspond roughly to the present Lisp "packages"; object-oriented organization of packages would help firm this up. Method categories do not exist and would have to be added (probably as comments in a standard form). The facilities of the Browser are simply another program, probably quite easy to write using DLISP. Note that Masterscope, with its more extended idea of "relatedness", could potentially enable a much more powerful browsing facility.

>>Integrated on-line documentation

HELPSYS is the best on-line reference documentation system available anywhere that we know of. Its drawback is that maintenance of the documentation is a tedious, non-incremental process. It will also need to be re-implemented for the Dorado.

>>Immediate incremental compilation

A very fast, simple, non-optimizing version of the byte compiler already exists. Our first approach will be to divide up the real compiler in a way that reduces the amount of optimization without sacrificing capability, and then explore other alternatives if this turns out to be too slow.

Packages

>>Arbitrary-precision integers with smooth transitions

This can be done as a package, although it requires some modifications to the error handling machinery to field overflow properly, and potentially complicates in-line compilation of integer arithmetic.

>>Paragraph formatting (display and Press)

The existing TXDT package provides most of this for files, and DLISP provides it for the display. Press output is a modest-sized package.

>>Bitmap and display coordinate objects

DLISP provides some facilities for manipulating these objects -- unclear how much, since documentation doesn't exist yet.

>>Paragraph and picture editor

DLISP has some kind of paragraph editor. A picture editor will have to wait for a redesign of the display facilities in the personal-computer environment.

>>Windows

DLISP has them.

>>Menus

DLISP has them.

Other

>>Non-preemptive window behavior

The DLISP scheduler is not documented yet. It clearly has some ability to keep track of several windows all waiting for input, but the cross-window selection capability suggests that it requires more cooperation between windows than Smalltalk does.

Inter-Office Memorandum

To	PE-Interest	Date	October 9, 1978
From	Lampson, Levin	Location	CSL, Palo Alto
Subject	Mesa Response to Lisp Challenge	File	[Ivy]<Lampson>mesaresponse.memo

XEROX

We culled five major (read "potentially difficult to implement") facilities from the memos by Bobrow and Masinter:

1) Programs as data
2) Variable syntax analysis
3) Dynamic variable binding
4) Safe language (subset)
5) Automatic storage allocation

This memo summarizes the analysis presented at the September 21st meeting.

1) Programs as data.

The ability to treat programs as data requires three major facilities within the Mesa environment: (a) an S-expression-like representation of source programs, (b) a means of using atoms (in the LISP sense) uniformly throughout the environment, and (c) a universal pointer mechanism.

S-expression representation. This seems straightforward in principle, though we haven't looked at the details. We think that a representation other than parse trees should be explicitly defined, permitting the representation of parse trees to change as the compiler/language evolve.

Atoms. The symbol tables built by the Mesa compiler currently include a mapping between source identifiers and internal unique ids (atoms). At present, these mappings exist only on a per-compilation unit basis, implying that they will need to be merged into a global map when the compilation unit "enters the environment". A similar merge presently occurs inside the compiler when the source program accesses an included module (i.e., the symbol table for that module is opened and the identifiers (used by the source program) are remapped into the current atom table). A merge into a global atom table therefore seems feasible, at least in principle.

Universal Pointers. We will need pointers that carry the type of their referent with them. This in turn requires a scheme for universal type IDs, which do not currently exist in Mesa. It seems possible to extend the techniques used to ensure uniqueness of record types to accommodate general types. Run-time efficiency is vital, so the check that two type IDs are equivalent must be fast. We anticipate using the LISP "spacing" technique, which places all data structures of the same type within a well-defined region of the address space (e.g., each page allocated for such dynamic structures can hold objects of precisely one type). This scheme avoids keeping any type ID bits with the pointer, but makes it difficult to point inside a composite data structure with a universal pointer. Two possible remedies have been suggested: (1) an explicit type identifier immediately preceding the (sub)structure, or (2) an indirection mechanism in the data structure itself. In any event, access to statically allocated variables (i.e., those in local and global frames) via universal pointers will have to be prohibited, unless special steps are taken to prevent problems with garbage collection.

2) Variable syntax analysis.

LISP provides three opportunities for the programmer to affect the parsing/interpretation of input text: (a) at scanning time, (b) at compile time, and (c) at function call time. (a) may be accomplished by providing scanner escapes to user-written code, which will replace some appropriate substring of the current parsing context (i.e., the as-yet unclosed lists) with a piece of program in S-expression form. It appears straightforward to provide a facility comparable in power to LISP's readtables -- a more elegant mechanism might be desirable. (b) may be accomplished by more-or-less conventional macro expansion along the lines of the Mesa mechanisms for in-line procedures. (c) is deemed undesirable and will not be provided.

3) Dynamic variable binding.

Mesa, at present, can only support static binding of identifiers. To be able to bind a name to a variable in an enclosing *dynamic* context, we need to know what variables are to be found in each frame. Instead of the LISP default, we propose that variables to which dynamic binding may occur be declared as such at compile time. The compiler can then produce a table of entries of the form

atom -> <frame,type>

for each compilation unit. Such a table could reside in the code segment, with a bit in the associated global frame(s) to indicate its existence. Dynamic binding would then occur much in the same way the signal-to-catch phrase binding occurs (but more efficiently).

4) Safe language

By a safe language we mean here one in which a pointer value of a given type POINTER TO T is guaranteed to point to a region of storage containing a value of type T. We will also consider a relaxation of this condition in which certain *regions* of the storage may not be safe in this sense; as long as values stored in such unsafe regions do not themselves contain pointers to safe regions, the damage which can be caused is confined to the unsafe regions.

There are three ways in which a pointer variable *p* may acquire an improper value

1) Lack of initialization - in this case the value of *p* is just the random bit pattern which happens to be in the storage occupied by *p* when it is instantiated.

2) Incorrect computation - in general a pointer value can only be the value of some pointer variable (which could of course be a record component) or the result returned by a storage allocation. If pointer values can be computed in other ways, e.g. with LOOPHOLEs or their equivalents, or by arithmetic, then there is no guarantee that such values point to storage containing values of the proper type.

3) Premature deallocation - in this case *p* is pointing at a region of storage which once contained a value of type T, but is now being used for some other purpose.

If none of these bad things can happen, then induction tells us that the language is safe with respect to pointers. In addition, it is necessary to do bounds checking on array references, since otherwise every array index is as dangerous as a pointer. Finally, array descriptors require the same treatment as pointers.

Preventing uninitialized pointers is straightforward, by generating code to initialize all pointers in procedure frames when they are allocated, and in dynamic variables of record and array types when they are allocated. Mesa 5 will have language constructs for allocating dynamic variables, so it is known where to do the initialization.

Preventing incorrect computation requires excluding all the dangerous constructs from the language

a) Loopholes into a pointer type or a pointer-containing type (PC for short)

b) Computed or overlaid variants in which the variant part is PC

c) Pointer arithmetic

d) Use of UNSPECIFIED (this restriction is probably stronger than necessary)

e) What did I forget

Preventing premature deallocation has two aspects: dynamic (explicitly allocated) variables and stack variables. For the latter, a simple and reasonable rule seems to be to disallow the use of @ on local variables. As partial compensation, it seems desirable to introduce var parameters (call by reference); these can be thought of as pointers to stack variables which are *always* dereferenced, so that it is not possible for the called procedure to make a copy of the pointer which will outlive the call. We also need to enforce a restriction that var parameters are not permitted across process and coroutine calls, since in those cases there is no guarantee that the caller will live longer than the called procedure (again, this seems too strong, but it isn't clear how to weaken it).

For dynamic variables some support from an automatic deallocation mechanism seems essential, since otherwise there is no way to tell when the programmer says Free(*p*) whether other pointers to *p*↑ still exist. We see three possibilities

a) Reference counting. This could be added to Mesa easily enough, but the cost during execution is rather high, and it is probably not a good idea in view of the other alternatives.

b) Deferred freeing. The idea is to make a list of all the storage freed by the programmer, and at suitable intervals to invoke the scanning phase of a garbage collector and check that there are no outstanding references to the storage queued to be freed. At the end of the scan any storage which has passed this check can be actually freed.

c) Automatic deallocation, discussed below.

Quite possibly deferred freeing will be worthwhile, since it is fairly easy to implement and has substantially lower running costs than fully automatic deallocation. With plenty of address space there should be no serious objection to a long interval between the programmer's call of Free and actual availability of the storage.

Mesa currently has primitive and unsafe facilities for relative pointers, which there is a design to strengthen and improve. Relative pointers have several appealing properties, one of which is particularly relevant in this context

a) They allow an entire complex structure to be moved in storage, or between storage and a file, as long as all its contained pointers are relative to the structure.

b) They can save substantial amounts of space, especially as the address space grows and pointers become longer.

c) Misuse of a relative pointer can only damage data in the region it refers to. If none of the storage in this region contains any external pointers, then no global damage can occur. Embedded external pointers are still possible if they are treated as array indices, i.e. go through fingers. The advantage is that the programmer can forgo the safety precautions for such relation pointers without risking the integrity of the entire system. Of course a bounds check is still required (analogous to an array bounds check), but in many cases of interest it can be provided by storing the pointers in *n*-bit fields and making the age of the region 2^n; an especially nice value for *n* is 16.

5) Automatic deallocation

Our proposal is to have automatically deallocated (AD) types, and to use the Deutsch-Bobrow scheme for incremental garbage collection. This requires knowledge of the location and type of every pointer to an AD type, but the techniques are well-known and straightforward.

Inter-Office Memorandum

To	EPE working group	Date	October 26, 1978
From	P. Deutsch	Location	Palo Alto
Subject	XLisp/EPE plan	Organization	CSL

XEROX

Filed on: <Deutsch>xlisppla1.memo on Maxc1

As requested by the EPE working group, this memo contains a gross (~5 item) plan for how we would get from where we are to an EPE XLisp, including preconditions, and a statement of how we would use 'warm bodies' to help us get there.

Since the overall XLisp design has only been presented orally, there may be steps in the plan whose meaning is not clear. A memo is in preparation which summarizes the relevant features of the design.

Plan

XLisp can evolve from our current systems in a series of steps, with a working (although not stable) system at each step. Here are the major milestones:

1) Get Alto Lisp running on the Dorado. Major work on XLisp is not possible on any other hardware -- the Alto is too slow and Maxc is too small. In particular, we need a Dorado Lisp that uses the IFU and hardware map, and all of the Dorado's real memory. This almost certainly means converting the kernel from Bcpl to Mesa. These issues are addressed in separate memos.

The next two steps can be done in either order, but not in parallel.

2a) Change the system to use shallow binding and have invisible indirection for variables. These are necessary for the lexical scoping rules and closure capability of XLisp. This has some impact on the compiler and a major impact on the functions that handle the stack.

2b) Add dictionaries to the system, but not lexically-scoped closures. This replaces the LOCALVARS and LINKFNS capabilities of the current system. It has major impact on the compiler, interpreter, and record and file packages, and minor impact in many places such as the editor and stack functions. Changing the way macros and special syntactic forms are handled can be done later, but we want to eliminate error-driven interpretation early. This strongly affects CLisp and user-defined macros and NLAMBDAs.

3) Implement closures. This includes lexically scoped FUNARGs, and arbitrary closures like Mesa module instances.

4) Implement the type calculus (types as objects, type checking), but entirely on a dynamic basis -- no variant representations, and no compile-time checking. A good deal of this is in the Decl package already.

5) Implement compile-time type checking. This affects the compiler, interpreter, and in general any function that manipulates function definitions (e.g. GETD/PUTD, APPLY*).

6) Implement variant representations. This has major impact on the instruction set (and therefore the compiler), and on all functions that manipulate records or the stack.

The following steps are necessary eventually, but do not affect the basic functional capabilities of the system:

7) Implement an infix front-end. This requires writing (or stealing) a text-oriented editor and a prettyprinter, and interfacing them to the Lisp editor and file package.

8) Implement permanent pointers. This involves completely rewriting the file package.

Using volunteers

Our ability to use volunteers depends somewhat on the ordering of the above steps, and on the qualifications of the volunteers. In particular, doing step 7 early might mean we could get some volunteers who only really felt at home in Mesa syntax. Here is how we could use people to help with each step:

1) Converting the Alto Lisp kernel to Mesa is mostly a transcription job, and could be done by anyone who knows both Bcpl and Mesa with relatively little briefing required. The other people-resource issues in Alto Lisp are discussed elsewhere.

2a) The shallow binding conversion requires people familiar with Lisp, with Lisp implementation, and with the Alto Lisp internal data structures. 'Warm bodies' are unlikely to be useful.

2b) Adding dictionaries is a tricky job; much of the work on the compiler and record package could be done by anyone familiar with Lisp, but much of it could only be done by the implementors of the programs in question.

3) Lexically scoped closures are a small task if done in a straightforward way. Any PARC Lisp programmer could do it. We know from SCHEME (and even from GEDANKEN) that Mesa-style instances can be built on top of lexically closed FUNARGs.

4) Implementing the type calculus requires understanding it, but little else. Whether we build on existing Decl code depends mostly on who does the work.

5) Implementing pre-execution type checking requires access to global knowledge of the Masterscope variety. Getting it done right probably requires an experienced person with an ongoing interest in type systems and Masterscope-like issues.

6) Variant representations are also tricky enough to require an experienced and dedicated person.

7) Anyone can write a decent parser and prettyprinter, and an adequate editor.

8) Permanent pointers, as part of a programming system, require some research. Integrating them into the system will be an ongoing project. The basic design, and an implementation of the lowest level operations for using them, can be done by anyone experienced.

XLisp: a Lisp answer to the EPE language challenge

- What are our goals?

- What technical methods will we use to achieve those goals?

- How can we achieve the goal in a step-by-step way?

- What do we need (in time and resources) to put this plan into effect?

Goal: upgrade (Dorado, Inter-)Lisp to include important EPE attributes as listed in the EPE report

We need to retain:

- Totally automatic storage management
- Easy program manipulation of programs
- "Instant" turnaround for program changes
- Lists and symbols as primary objects
- Runtime type system

We aim to add:

- Type system with static checking capability
- Abstractable interfaces
- Improved runtime efficiency
- Scope-oriented protection
- Consistent compilation, version bookkeeping
- Uniform notion of calling, instantiating, closures

Method:

Define system entirely in terms of dynamic objects -- name scopes, types, etc. are objects

Static checking & optimization come through added declarations

Dictionaries for name interpretation -- unifies notions of argument list, record, interface

Maps names to attributes such as type (abstract & representation), protection, manifest value, etc.

Protection is associated jointly with the dictionary and the path from the accessor

Dictionaries describe how to create instances of records

Calling = instantiating (the frame) + copying (the arguments) + running

Closure formation = instantiating + copying

Coroutine transfer = copying + running

Type system includes ideas of record formation (via dictionaries), specialization by adding attributes, extension (subclassing), multiple representations

Coercion between representations (e.g. tagged vs. untagged) or degrees of specialization is automatic

Checking is done with EQ (identity)

Permanent pointers extend the EQ concept to the file system

Plan:

1) ****** Get Alto Lisp running on the Dorado
(more about this next week!)

2) ***** Add dictionaries for scoping only, not generalized closures

3) *** Add generalized closures
(at least one implementation of this already exists)

4) ** Add a type calculus, but do all checking at run time
(existing Decl package does a lot of this)

5) **** Add compile-time type checking

6) *** Add variant representations (e.g. unboxed numbers)

7) * Add an infix (Mesa-like) front end

8) ?? Implement permanent pointers

Resources:

Think of * = 3 person months

All but (7) and perhaps (8) require experienced people who understand the system implementation

We think critical mass is 4 essentially full-time professionals

Providing Lisp strengths in Mesa

There are two levels

- Upper: Integrated editor
  - Masterscope
  - Programmer's assistant
- Lower: Programs as data
  - Variable syntax analysis
  - Dynamic variable binding
  - Safe language
  - Automatic storage deallocation

- Programs as data
  - Atoms
  - Standard S-expression representation
  - Universal pointers
    - Unique type IDs
    - "Spacing"
    - Discrimination by CASE or by assignment
- Variable syntax analysis
  - At READ (scanner) time (read tables)
  - At compile time (syntax macros)
  - At execution time -- not in Mesa

Dynamic binding

- Needs a table which maps
  - VariableID => (location in frame, type)
- Mesa already does this for signals

Safe language

Pointers into regions of storage which contain system data structures or other such pointers must be handled correctly.

There are three problem areas:

- Initialization
- Computation -- no LOOPHOLE, UNSPECIFIED, computed or overlaid variants involving pointers, or pointer arithmetic; bounds checks for array references
- Dangling reference -- forbid deallocation

Relative pointers to data which doesn't contain absolute pointers need not be safe

Automatic deallocation

Use the Deutsch-Bobrow incremental collector, as in Alto/Dorado Lisp

Symbol table information

Atoms -- print name <= => internal ID

Name => (location, type) map for frames

Unique type identifiers

Type => (location, type) map for pointer-containing types

Inter-Office Memorandum

To	EPE working group	Date	November 7, 1978
From	P. Deutsch	Location	Palo Alto
Subject	XLisp/EPE plan, revised	Organization	CSL

XEROX

Filed on: <Deutsch>xlispplan2.memo on Maxc1

As requested by the EPE working group, this memo contains a copy of the 8-step plan presented in last week's Dealer for how we would get from where we are to an EPE XLisp, including preconditions, time estimates, and a statement of what we would have at each step. This memo supersedes the previous memo <Deutsch>xlispplan1.memo of October 26.

Plan

XLisp can evolve from our current systems in a series of steps, with a working (although not stable) system after each step. The steps listed below need not be done in exactly the order given, although I believe this order leads to the least total work. I also believe little parallelism is possible *between* any two of the steps, although, as my assignments of people indicate, a lot of parallelism occurs *within* a single step. The names attached to steps other than the first should not be taken too literally -- the named people have not been consulted.

The total of the figures below is: 87 work months, 27 1/2 elapsed months.

1) Dorado Interlisp
Work months: 18
Elapsed months: 6?

People: Bobrow, Deutsch, Haugeland, Teitelman; Birrell, Fikes, Goldstein, Kaplan, M. Kay; Laaser, Masinter, Thompson

Get Alto Lisp running on the Dorado. Major work on XLisp is not possible on any other hardware -- the Alto is too slow and Maxc is too small. In particular, we need a Dorado Lisp that uses the hardware map, and all of the Dorado's real memory. Danny Bobrow has written a detailed memo on the subject of getting a Dorado Lisp up, including assignment of people to subtasks.

After step 1 we have a usable Interlisp system running on the Dorado.

1') Shallow binding
Work months: 3
Elapsed months: 1

People: Bobrow (stack fns & interpreter), Deutsch (compiler & loader), Haugeland (microcode)

Change the system to use shallow binding and have invisible indirection for variables. In addition to performance improvements, these changes help support the lexical scoping rules and closure capability of XLisp. This has some impact on the compiler and a major impact on the functions that handle the stack.

Step 1' produces no externally visible changes in the system, except that GLOBALVAR declarations are no longer necessary.

This step can actually be done at any time; it is easiest to do here, when less code will need to be changed. It requires people familiar with Lisp, with Lisp implementation, and with the Alto Lisp internal data structures.

Step 1' may not be necessary at all if microcode support for dynamic binding proves fast enough. I am inclined to think this will not be the case, especially if GLOBALVAR declarations are eliminated.

1") Clean up macros
Work months: 6
Elapsed months: 1

People: Bobrow (interpreter), Deutsch (compiler), Teitelman (DWIM/CLisp), any 3 Lisp programmers (scan the system); defer work on Masterscope

Change the way macros and special syntactic forms are handled, to eliminate "error-driven interpretation". In particular, eliminate DWIM corrections that require altering or examining the state of the interpreter. This strongly affects CLisp and user-defined macros and NLAMBDAs. Relatively small changes are also required in the compiler, interpreter, and Masterscope.

The current MACRO scheme does not fit into the scoped view of the world, and has caused many obscure bugs; NLAMBDAs make separate MACRO definitions necessary, and require internal EVALs, which we would like to eliminate since they will require elaborate run-time checking. We can arrange things so that after step 1" the system will still run existing Interlisp programs, but MACRO properties and NLAMBDAs will produce warning messages. On the other hand, packages like the iteration and pattern match statements will fit into the system much more cleanly.

This too could be done later, but it makes things a lot simpler if the system doesn't have to support run-time DWIM correction while dictionaries are being installed. Any good Lisp programmer can go over all the system code to change NLAMBDAs and macros to conform to the XLisp scheme; only Warren can make the necessary simplifications in DWIM and CLisp.

2) Dictionaries
Work months: 15
Elapsed months: 2 (design), 3 (implementation)

People (design): Bobrow, Deutsch, Masinter, Horning/Lampson

People (implementation): Bobrow (stack & interpreter), Deutsch (compiler), Fikes (record package), Kaplan (dictionary objects), Teitelman (file package & miscellaneous); defer work on Masterscope

Add dictionaries to the system, but not lexically-scoped closures. This replaces the LOCALVARS and LINKFNS capabilities of the current system. It has major impact on the compiler, interpreter, and record and file packages, and minor impact in many places such

as the editor and stack functions. The hardest part of this step is probably designing the proper interfaces to dictionaries-as-virtual-objects.

After step 2 we have a system which supports localization of names. Individual packages can be rewritten to take advantage of this at any time. Some lame-duck code must remain in the file package to be able to dump and load individual definitions in a global dictionary so that existing programs will run.

Dictionaries are the largest single change to the system, requiring substantial design work. Some of the work on the compiler and record package could be done by anyone familiar with Lisp, but most of it could only be done by the implementors of the programs in question.

Steps 2 and 3 can be done in either order, but the presence of dictionaries, which guarantee that most names are only known within a limited lexical scope, makes it possible to do much better optimization on closures. See Guy Steele's thesis on RABBIT (a compiler for SCHEME, a lexically scoped Lisp dialect) for details.

3) Closures
Work months: 9
Elapsed months: 1 (design), 2 (implementation)

People (design): Bobrow, Deutsch, Levin

People (implementation): Bobrow (stack functions), Deutsch (compiler), Haugeland (microcode), Teitelman (BREAK et al)

Implement closures. This includes lexically scoped FUNARGs, and arbitrary closures like Mesa module instances. This affects all programs that deal with the representation of dynamic state, such as the stack functions and the BREAK package.

After this step the system will support instantiable packages and Smalltalk-style object-oriented programming. Old programs are not affected; however, some existing packages should be recast in instantiable form, and there will be a period of confusion and trauma while this happens.

Careful design is needed here to produce reasonably efficient results. We will probably want to change the instruction set at some point to make record access faster. Implementation wizards, including someone who really understands the Mesa and Smalltalk way of doing things, are needed for this step.

4) Type calculus
Work months: 9
Elapsed months: 1 1/2 (design), 2 (implementation)

People (design): Bobrow, Deutsch, Horning/Lampson, Scott?

People (implementation): Bobrow (interpreter), Deutsch (compiler), Kaplan (type objects)

Implement the type calculus (types as objects, type checking), but entirely on a dynamic basis -- no variant representations, and no compile-time checking. A good deal of this is in the Decl package already.

The result of this step is to make the type system a first-class citizen, rather than an intermittent visitor as in the Decl package. Programmers can start writing type declarations in the knowledge that they will reap the benefits after step 5.

Getting the type calculus right will require help from people with particular insight in this area such as Horning and Scott. Meshing the type system with the compiler, interpreter, and Masterscope is not believed to be a very big job, given the Decl experience.

Note that in the absence of permanent pointers (step 8), some method involving unique type IDs recorded in compiled code files will be required to do checking properly, as in Mesa. Modulo this question and the related question of protection, most of step 4 could be done in any order relative to steps 2 and 3.

5) Compile-time type checking
Work months: 12
Elapsed months: 3

People: Deutsch, Levin, Satterthwaite (compiler), Kaplan (PUTD, APPLY*, etc.)

Implement compile-time type checking. This affects the compiler, interpreter, and in general any function that manipulates function definitions (e.g. GETD/PUTD, APPLY*).

This step gives us a system that has nearly all the functional capability of a "safe" Mesa, although it is considerably less efficient in its representation of some kinds of objects, e.g. bounded integers and set elements. It is not clear what to do about the Binder.

Getting pre-execution type checking done right probably requires an experienced person with an ongoing interest in type systems and Masterscope-like issues. A hard issue we have not examined is just how simple we can make the part of the checking system that must be absolutely correct: for example, it would be nice if it didn't need to include all of Masterscope.

6) Variant representations
Work months: 6
Elapsed months: 1 (design), 1 (implementation)

People (design): Deutsch, Satterthwaite

People (implementation): Bobrow (stack functions), Deutsch (compiler), Fikes (record package), Haugeland (microcode)

Implement variant representations. This has major impact on the instruction set (and therefore the compiler), and on all functions that manipulate records or the stack.

This system should be competitive with Mesa, aside from the questions of LOOPHOLEs, pointer arithmetic, etc., and of course it has a lot of capabilities not in current Mesa.

This step, like the previous one, requires someone with compiler experience and careful scrutiny to see how well we can isolate the code that must be absolutely correct.

The remaining steps are necessary eventually, but do not affect the basic functional capabilities of the system.

7) Infix front end
Work months: 3

Elapsed months: 1

People: Goldstein (prettyprinter), Swinehart (editor), Teitelman (integration)

Implement an infix front end. This requires writing (or stealing) a text-oriented editor and a prettyprinter, and interfacing them to the Lisp editor and file package.

Anyone can write a decent parser and prettyprinter, and an adequate program editor. This step could be done much earlier, and might help us get people who like Mesa syntax to work on the system.

8) Permanent pointers
Work months: 6?
Elapsed months: 2?

People (design): Deutsch, Horning, Sturgis

People (implementation): Sturgis (primitives), Teitelman (file package)

Implement permanent pointers. This involves completely rewriting the file package.

Permanent pointers, as part of a programming system, require some research. Integrating them into the system will be an ongoing project. The basic design, and an implementation of the lowest level operations for using them, needs someone who understands the considerations that go into file systems like Juniper.

Inter-Office Memorandum

DRAFT - DRAFT - DRAFT - DRAFT

To	PE Group	Date	November 9, 1978 1:09 PM
From	Deutsch, Bobrow	Location	Palo Alto
Subject	XLisp design -- a detailed response to the Mesa challenges	Organization	CSL

XEROX

Filed on: <Deutsch>xlisp1.memo on Maxcl

****** Note: this is a document-in-progress. It is not stored in the PE file on <Deutsch>pe2>. Always check the date and time in the "Date" field above to make sure you have the latest version. ******

Introduction

This document describes part of the design for a system we are calling XLisp. It is a dialect of Lisp, based largely on SCHEME (a lexically-scoped Lisp dialect developed at MIT), and adding many capabilities derived from Mesa and other modern statically typed languages. The goals of XLisp are roughly the following:

> Keep the current ability of Lisp to manipulate programs and to carry out computations with no type or scope declarations required, at approximately current efficiency.
>
> Add the ability to achieve greater efficiency and statically checkable protection, by adding declarations.
>
> Develop a conceptual framework for thinking about scope and type issues which has some room for future exploration of capabilities not present in either Lisp or Mesa.

The purpose of this document is not to exhibit a design that we think we could start to implement tomorrow: a lot of details are missing. Our purpose is rather:

> To demonstrate to the Mesa-oriented community that we really know how to achieve the valuable capabilities of Mesa within Lisp, and that our view is not only implementable but conceptually clean (hopefully, cleaner than current Mesa).
>
> To persuade the Lisp-oriented community that the changes they will have to make in the way they do things are a small price to pay for the new power they will obtain through this design.

This document needs, but currently lacks, a tremendous amount of motivational and explanatory material (including examples). One kind of material is needed for Lisp-oriented readers to answer the question, "Why is this changed?" Another kind is needed for Mesa-oriented readers to answer the question, "What Mesa challenge does this respond to?" In this edition of the design, because of time pressure, we have opted for a presentation of the end result with little indication of how we got there.

Organization of this document

This document is not organized properly yet. The right order of presentation is probably:

Types excluding records

Record types and their dictionaries

Interface types and their dictionaries

Function objects and environments

Miscellaneous

Types

****** This section is still confused. I- and D- types have come and gone at least twice. There is a section on declarations, assertions, and invariants near the end of the memo which indicates some of the source of the confusion. ******

Framework

Types are objects, just like integers or lists. There are three kinds of types: interface or I-types, which describe how objects behave in general; descriptive or D-types, which specify predicates that hold true of objects in particular situations; and representation or R-types, which describe the bit patterns used to represent objects inside the machine. Conceptually every object carries its I-type with it at execution time, but not its R-type (D-types are associated with variables, not with objects). The system provides a default representation for every I-type. The phrases "X is of type T" and "X satisfies type T" are interchangeable -- I- and D-types are predicates.

I-types are built up using the following operators (the names and Mesa-like syntax are for expository purposes only):

ANY[] is a type which all objects belong to.

FLOAT[] is a type for floating point numbers of one particular precision and range defined by the "hardware".

ATOM[] is a type for atoms.

INTEGER[i1,i2:IntegerOrNIL] is a type for objects which can take on integer values from i1 through i2 inclusive. NIL means there is no limit on that end.

CONSTANT[x] is a type for objects which can take on only the value x. This is not as useless as it sounds: for example, NIL occupies a special place in the type space.

SATISFIES[p:FUNCTION[[ANY[]],[Boolean]],t:Type] is a type for objects x of type t such that p[x] is true.

ANYOF[t1 ... tn:Type] is a type for objects which can be of any of the types t1 ... tn.

ALLOF[t1 ... tn:Type] is a type for objects which must be of types t1 ... tn simultaneously.

READONLY[t:Type] is a type for objects which are of type t shorn of any operations which produce externally visible side-effects (more on side-effects below).

UNIQUE[t:Type] is a type which is the same as t, except that objects of this type are distinguishable from objects of type t and of any other type created by UNIQUE[t]. This operation is sometimes called "painting".

FUNCTION[argT,resultT:Type] is a type for functions which take arguments of type argT and return results of type resultT. ArgT and resultT must be structure types.

VARID[t:Type] is a type for variable id's of type t. Variable id's are used to link together binders and users of variables, leaving the identity of the variable unbound. (Same idea as the Mesa SIGNAL linking mechanism.)

Structure types are discussed in a following subsection.

[How do variant R-types get defined??]

Declarations

An S-type declaration just establishes an invariant for a variable -- checked whenever the variable is stored into, can be assumed whenever the variable is used. S-types can also be used as predicates, and wrapped around expressions. R-type declarations specify the representation of a quantity and hence are only meaningful for variables, as opposed to arbitrary expressions. ("Variables" include record fields, of course.)

Structure types

The other kind of definable S-type is a structure type. It has named components, each of which may have type declarations. There are two kinds of components: fields, which obey certain rules set forth below that make them act like variables, and operations.

Every field component has two associated operations called fetching and replacing that field. These operations obey the following rules:

(1) Fetching a field has no externally visible side-effects (although it may have internal side-effects like caching or statistic-taking -- obviously "visible" can't be defined rigorously). In particular, it does not affect the contents of any field.

(2) Replacing a field has no externally visible side-effects on any operation other than fetching that same field.

(3) Fetching a field always returns what was last replaced into that field.

(4) No operation other than replacing a field affects what fetching that field will return.

The default R-type for fields is a storage cell: if the programmer specifies a different R-type, its adherence to the above rules is up to him.

Fields in structure types may have names, or an ordering, or both. The ordering is for argument and constructor lists -- it has nothing to do with storage format.

Arrays are a structure type in which the fields, instead of having names, are indexed by a set which may be of any type. If the type is an integer range, the default representation is a traditional array; if not, the default is a hash array.

An interface type, in the Mesa sense, is a structure type with fields which are of various function types. A Mesa module is a group of functions which all agree to use the same instance(s) of the interface types they import. Making an instance of a module consists of making an instance of the interface type it implements, filling in the fields with closures of the exported functions in which the imported interface pointers get bound.

****** Fields require an instance. Smalltalk operations require an instance. Interfaces don't necessarily require an instance. How to resolve this best? Mention "fetch", "replace", "perform"

syntax for things that need an instance.

Conformity and coercions

Type checking depends only on the S-types of the objects. The conformity relation is straightforward to compute, given the type composition primitives. The compiler does all it can, and then produces code to do the remainder of the check.

Coercions are meaningful at both the S-type and R-type levels. S-type coercions up or down the type lattice are mostly either no-ops or type checks, and can be supplied automatically. Other S-type coercions require user-supplied functions. R-type conversions require user-supplied functions to convert to and from the default R-type, the familiar Lisp typed pointer.

Here are the coercions available among S-types. The notation t1 -> t2 means that an attempt is being made to convert a quantity U of type t1 to a quantity V of type t2.

t -> ANY[]: no checking required
ANY[] -> t: requires a type check

FLOAT[] -> INTEGER[i1,i2]: may require a range check
INTEGER[i1,i2] -> FLOAT[]: always allowed

CONSTANT[x] -> t: requires checking that TYPEP[x,t]
t -> CONSTANT[x]: requires checking EQUAL[U,x]

INTEGER[i1,i2] -> INTEGER[j1,j2]:
If i2<j1 or i1>j2, not allowed.
If j1≤i1 and j2≥i2, allowed with no check required.
Otherwise requires a range check on U.

SATISFIES[p,t] -> t': use algorithm for t -> t'
t -> SATISFIES[p,t']: p[U] must be true, and t -> t' allowed

ANYOF[t1 ... tn] -> t: for some j, must have TYPEP[U,j] and tj -> t allowed
t -> ANYOF[t1 ... tn]: for some j, must have t -> tj allowed

Functions, closures, bindings, and environments

Function and environment objects

Current Interlisp encourages the user to confuse function names, list structures representing functions, and function objects, but does not treat them uniformly: for example, the first argument of APPLY* may be a function name or a list structure, while the second argument of PUTD may be a list structure or a compiled or SUBR function object.

In XLisp we distinguish between code objects, which are merely different forms of program text, and environment objects, which may contain values for some or all of the variables of the text. There are three kinds of code objects: primitive (representing SUBRs and microcoded functions), compiled, and interpreted functions. Conceptually, every code object contains a pointer back to the naming environment (called "scope" below) in which it is embedded: an interpreted code object is thus not simply an S-expression, but rather a pair <S-expression, scope>.

In XLisp the only thing that can be executed is an environment object, or EO: EOs, and only EOs, are legal as function definitions. An EO is either a code object or (recursively) an EO plus bindings for some of its unbound variables. The state of an EO logically always contains two components:

What kind of data the environment expects when control next arrives: none, a particular

argument list, or an operation id and associated arguments. Note in particular that the continuation, i.e. the identity of the calling EO, is an argument.

Whether a new instance of something needs to be created when control arrives. (A new instance is always created if the environment expects an <operation, arguments> pair.)

Naturally, when an EO transfers control outside itself, it specifies what kind of data it is passing to the new EO: none, an argument list, or <operation, arguments>. If the sender does not pass itself as the continuation, then (in effect) it has lost one reference to itself, and if there are no more references it can be freed. An example of this is a function return.

A traditional Lisp function is an EO which expects an argument list and creates a frame to hold it. A closure (more on these below) is an EO which already has one or more frames as part of it. A Smalltalk instance or Mesa module instance is an EO which expects <operation, arguments> and creates an appropriate frame.

Some kinds of EO cause errors if an attempt is made to execute them with any of their variables unbound. For EOs that do allow such execution, then an error occurs if an attempt is made to actually reference an unbound variable. Clearly we will want several different representations of EOs depending on how much and what kind of state information they include, corresponding to different kinds of functions, closures, and environments.

Closures

****** Need to talk here about local and fluid variables; how interfaces get hooked up; various kinds of instantiation (Mesa & Smalltalk modes). How VARID types with the extra level of indirection do SIGNAL binding and Lisp dynamic variables.

Given the existence of fluid variables, there are two different kinds of state one might wish to capture in a closure: the current bindings of the lexically enclosing variables, and the current fluid binding environment. An XLisp closure always captures the first of these (but only the bindings for those variables actually used free within the function being closed -- this is determinable statically); optionally, the programmer can specify whether the fluid binding environment is to be captured as well. (Individual fluid bindings can be captured by rebinding them to local variables, and then rebinding the fluid variables to the captured values inside the function -- of course this can be done with a macro.)

Syntax

In current Interlisp, LAMBDA and NLAMBDA play an anomalous role as syntactic markers, different from any other syntactic context. XLisp follows the SCHEME convention: NLAMBDA does not exist, and LAMBDA is a *function* which takes an argument list and a body and returns a function object. This leads to few changes from the user's point of view: the main one is that the standard representation for a functional argument becomes simply (LAMBDA --), although (FUNCTION (LAMBDA --)) can still be acceptable by making FUNCTION a macro.

The form (NLAMBDA var . forms) can be converted to a MACRO with the definition

```
(LAMBDA (FORM)
   (SUBPAIR 'REST (CDR FORM)
      '((LAMBDA (var) . forms) (QUOTE REST)) ))
```

and similarly for NLAMBDAs which spread their arguments, but this leads to very inefficient execution. There is no way to duplicate Interlisp's handling of NLAMBDAs exactly, since MACROs are not functions and cannot be APPLY'ed or APPLY*'ed. We see nothing wrong with this.

****** Need to do something about multiple return values. Maybe (RESULTS form var-for-val2 ... var-for-valn)?

Protection

Mesa provides four kinds of protection that are not checkable at compile time:

1) Variables, unless otherwise declared, are not accessible at run time by programs outside the lexical scope where they are defined.

2) A client and an implementor must use identical (EQ) versions of an interface.

3) An implementor can define types whose contents are (partially) private, but which the client can use to declare variables.

4) The types of the arguments passed in a control transfer always agree, regardless of how the environment receiving control was obtained (e.g. by binding, through a PORT, through a procedure variable).

All of these are based on intra-module checking by the compiler, and inter-module enforcement based on (2). This, in turn, is accomplished by creating a unique name for an interface when it is compiled, and copying that unique name into both the client and the implementor when they are compiled.

Our type checker provides (2) by simply using EQ -- XLisp always keeps with any compiled object a pointer to the dictionaries accessed during compilation, which include all imported and exported interfaces. The permanent pointer mechanism described under Miscellaneous below guarantees that the EQ relation is correct even when externally filed data are involved.

(4) is also a type checking question. If the user declares procedure and environment variables appropriately (which is automatic if an interface dictionary is involved), then the check can be made when the interface is bound rather than at the time of the control transfer.

The privacy information in dictionaries and scopes provides (1), since the only way to access a non-local variable is by going from a frame or other object to the dictionary that describes it, and the operations to access objects in general are implemented within dictionaries and check the privacy information. For programs like editors that must be able to make changes in dictionaries, we provide a way to make a less-protected (but non-EQ) copy of a dictionary, and an operation that replaces the entire contents of a dictionary with that of another, rejecting it if it does not satisfy appropriate protection requirements. This operation also decides how much current state is invalidated by the change.

The heart of our protection system is (3). Something like (3) is required to be able to have virtual objects, and objects accessible only through interface procedures (operations). In XLisp one can declare a record type as having sealable instances. What this means is that the record has a piece of code associated with it, and a "sealed" flag. If the record is in unsealed state, anyone may access it freely. If the record is in sealed state, only the associated code may access it. A record must be sealed to be used as the local environment of a function: the operation of calling a function logically breaks down into the steps of creating an instance of its local frame type, filling the parameters into the instance, sealing the frame, and then running the function. Similarly, records which define Smalltalk-like instances are sealed in a way that accepts incoming messages and then executes the implementing function. Such functions can access the sealed record by having a link in their scope structure that says they implement an operation on that record. (This is no less safe than the IMPLEMENTING construct in Mesa.)

A sealable record and its underlying unsealable record are two different types. Coercing the former to the latter (going into an implementor) requires that the sealed flag be set. Coercing the latter to the former is always possible -- the result is always sealed. The implementor can specify an arbitrary error check to be applied to a sealable record in order for the seal to be set: the check and the sealing take place atomically. This can be accomplished by copying the record, doing the check,

and then atomically copying the checked result back. Alternatively, the record can be accessed through a level of indirection, and the indirect pointer can be disabled at the start of the check.

Dictionaries

A new kind of object called a dictionary is central to the interpretation of names (and, hence, programs) in XLisp. A dictionary consists of one or more name tables, each of which applies to a different syntactic context (currently, only "function" or "variable") and specifies the interpretation of a set of names in that context: each entry in a name table consists of a name (litatom) and an associated property list. We have already discussed dictionaries above -- they are what defines a structure type. Every function has an associated dictionary whose fields are the function's arguments (functions of an arbitrary number of arguments have numbered, rather than named, fields).

****** What's on the property list?

Names vs. atoms

In current Interlisp, global values and function definitions are associated with litatoms. In XLisp, they are associated with <name, dictionary> pairs, where the name is a litatom. For fluid variables known globally, the distinction can be ignored, but the functions to manipulate values and definitions (GETD, GETATOMVAL, etc.) will all be redefined or extended to take an environment (scope plus execution environment) as argument.

Scopes

Conceptually, a scope is an area of a program within which the interpretation of names does not change; implementationally, it is an object which specifies how to determine the meanings of names. We will freely confuse the concept and implementation in the discussion that follows.

A scope is made up of components, each of which gives the meaning of some names, and a rule for deciding how to resolve conflicts and what to do with undefined names. Each component may be a scope itself, or it may be a dictionary, which actually associates names and meanings.

Scopes are built up and modified in four ways: by declarative information that is part of the program text; as a side effect of certain program constructs such as LAMBDA and PROG; by editing or otherwise operating on the program; and by direct calls on the functions that operate on the representation of scopes. [How do we confine the last of these to safe uses??] Scopes are not a "compile-time" phenomenon -- they apply equally to interpretation, analysis, and all modes of compilation -- but compilation can be viewed as a process of binding scope-derived information into the code.

Every name in a program must eventually resolve to a meaning of one of three basic kinds: manifest, in which the actual value is found in a dictionary (for example, local and system functions, record names, and what are now thought of as compile-time constants); dynamic, in which the value is found in a run-time data structure (for example, most variables and non-local functions); or unbound, in which only declarative information (if that) is available. Unbound names must be bound before a function using them can be compiled, or before a reference to one can be completed successfully during interpretive execution; however, editing, printing, analysis, and interpretation of other parts of the function are possible even in the presence of unbound names.

In principle, every dynamic name eventually reduces to a value obtained by applying elementary operations (at roughly the instruction set level) to constants and to pointers which form part of the virtual machine's state, such as the current stack frame pointer. Thus, in principle we only need one kind of name table entry: a way to macro-expand a name into a form eventually containing only constants (since ultimately the primitive operations are represented by constant function objects). An appropriate set of primitive operations would be:

(FLUIDREF fluid-cell) -> field-reference
(LOCALREF variable-number [frame-pointer]) -> field-reference
(FIELDREF field-descriptor [base-pointer]) -> field-reference
(FETCH field-reference) -> value
(REPLACE field-reference value) -> value
(ENCLOSING [frame-pointer]) -> frame-pointer (the lexical access link)
(CLOSURE function-object var1 ... varn) -> function-object
(APPLYFN function-object fluid-binding-link arg1 ... argn) -> value

(A field-reference is a new kind of reference object which can be used for either fetching or storing.) Indeed, these primitives are available in the system, since they are needed for the interpreter and compiler, but their use is deliberately rendered inconvenient (except for APPLYFN) since it can lead to violations of scope rules on which not only compiler optimizations but the correct working of programs may depend. (The mechanism for isolating the use of these primitives, like the even lower-level primitives like CLOSER, is to install the corresponding primitive function objects, not in the global dictionary, but in a private dictionary. Scopes using these operations must mention this dictionary explicitly.) As is usual in implementing protection systems, we provide, instead of these primitives, a set of higher-level constructs that cover all programming needs above the lowest level of system implementation, which lend themselves better to static analysis and permit more enforceable protection boundaries.

Here, then, are the constructs actually provided in the form of different kinds of name table entries:

(MANIFEST value) -- the actual value is present in the name table. The value may be any Lisp object (data structure, function object, etc.). The value is treated as a constant: changing it may require recompiling any function that uses it. If it is a non-scalar value, it is only given out in read-only form, and editing requires copying the affected part and then recopying when installing in the name table.

(SCRATCH value) -- the value is present in the name table, but it is not read-only. One can think of this as an OWN structure in the Algol sense.

(MACRO function-object) -- the name table contains a function which is applied to the form (application or atom), and the result is taken in place of the original form. It is vital that macros produce consistent results, i.e. only depend on the form being expanded and the scope in which it is being expanded. We assert that is sufficient to analyze the macro function and all functions it calls to make sure that they refer only to local and manifest variables, and to pass the macro function its arguments in read-only form; then any operations with side effects, if applied to objects not created during this call of the macro function, will cause an error.

(FLUID fluid-cell) -- this is the traditional Lisp binding method. Note that function and variable names are different for fluid binding purposes, and that the name itself is not a global litatom but an object local to the dictionary, so that the binder and the user of a fluid name must share the dictionary.

(UNBOUND)

(LOCAL location-in-frame) -- this is the SCHEME lexically bound variable. It differs from a fluid variable in that closures which use local variables free retain a shared pointer to the binding in effect at the time the closure is *created*, while a free use of a fluid variable refers to the binding at the time the closure is *applied*. (See the discussion of Closures under Functions, above, for more detail.) LOCAL names are also used for fields in records: in this case, location-in-frame defines the location of the field within the record instance.

(RELATIVE name scope base) -- this is used to refer to objects found in other scopes. Examples include argument or outer PROG variables referenced from within a PROG, instance variables and abstract operations in a record, and individual entries (usually manifest or fluid) imported from other scopes. A relative entry contains three parts: a

name, a pointer to the scope in which to interpret the name, and, if the interpretation is local, an expression (interpreted in the original scope) which gives the base pointer. This base pointer becomes the default base pointer for the purpose of the interpretation, so cascading of relative entries will work properly. The Mesa OPEN construct amounts to making appropriate RELATIVE entries in the current scope for all the names in the object being opened.

Miscellaneous

Pointers

In current Interlisp, the only way to protect a structured object which the system allows the user to access is to copy the object when it enters the protected area and copy it again on any request to see it. We choose to provide a more efficient mechanism that has many other uses as well, namely, to introduce a read-only bit into pointers: this still requires copying going in, but not going out. The read-only bit is propagated, in that any pointer retrieved through a read-only pointer is made read-only.

The MIT Lisp machine uses a combination of shallow binding and "invisible indirection" to achieve its efficient implementation of lexical scoping and correct semantics of closures. We intend to use the same technique. This requires being able to specify in a local variable binding in a stack frame that a fluid cell is to be used instead, which is a limited form of invisible indirection. We don't currently plan to provide this in other contexts.

Syntax

The syntax for programs is ultimately an S-expression syntax. There are two ways that this syntax gets extended:

There will be a preprocessor that converts an infix notation into S-expressions, and a prettyprinter that inverts this process, so that corrections made to the S-expressions can easily get reflected in the infix source. We don't have any prejudice about which notation programmers will prefer to use -- both will be supported for listings, editing, etc. Programs, however, will always operate on the S-expressions.

Macros can produce alternative syntax at the S-expression level. AND/OR, PROG, SELECTQ, and CLisp constructs like the iterative statement are good examples of this. Some such constructs (like the iterative statement) are complex enough to be packages in their own right. However, such alternative syntax is *always* introduced by a well-defined (macro) name in the function position. We may support the CLisp technique of caching the translation of such constructs, to speed up interpretation.

We feel that CLisp-style "error-driven interpretation" is a mistake: the language should be defined in such a way that if an execution-time error occurs, it is the result of an erroneous computation or source program, not of a construct that the user thinks of as valid but that just happens to get executed through the error machinery.

Even in present Lisp, "program syntax" is more than the syntax of expressions -- it really includes DECL constructs, BLOCKS declarations, fileCOMS syntax, etc. ****** We haven't yet really dealt with the question of syntaxes for defining linkage and dictionary construction.

Filing

External filing systems provide capabilities (e.g. backup, archiving, scavenging, sharing, large capacity, very high reliability) that we do not understand how to provide entirely within a programming system. XLisp programs contain inter-program references (to dictionaries, in particular) that must be representable on external files, so that a program knows it will have the

same environment when run subsequently, rather than some environment made up of objects that happen to have the same strings as their names.

We define a new object called a permanent name (PN) which consists of two parts: a file name (a string), which is to be considered only a hint, and a permanent id (PID, an integer of roughly 64 bits), which is the actual name of the object, and which is guaranteed different for every permanent object. Potentially, many different kinds of objects have permanent names -- function definitions, dictionaries, maybe any object whatever. To find the object addressed by a permanent name, it is necessary to hunt around in the file system, presumably starting at the file named in the PN, until an object with the right PID is found. Each file contains a directory (like the Interlisp filemap) that maps PIDs to text representations of objects in the file.

A PN is only needed for an object if a cross-file reference (or multiple reference within a single file) is made to it. PNs and PN directories are large, so we think it would be satisfactory to assign a PN only when (1) an object was written on a file, and "potentially referenced via a PN" was a property of its type; (2) a program asked for a PN for an object.

PNs are a basic mechanism for constructing permanent pointers. To make them useful, one needs such things as the notion of versions of an object, directories mapping PNs of original versions to PNs of current versions, etc. The crucial question is one of overwriting: how can one change an object without changing its PN? Fortunately most (all?) file storage systems provide some kind of atomic operation which is guaranteed to execute either completely or not at all, and an atomic overwriting operation can be constructed using this.

Declarations, assertions, and invariants

There seem to be two kinds of declarations/assertions:

> Those which are associated with *control*, i.e. which hold precisely when control reaches a particular point in the program.
>
> Those which are associated with *objects*, i.e. which hold throughout an object's lifetime independent of the flow of control.

Type declarations for fields are a kind of object assertion. They are easy to enforce, by simply checking whenever one stores into the field that the value being stored is of the right type. However, more complex invariants relating various fields of an object cannot be enforced this way. What seems to be needed for this case is a way to make updating operations apparently atomic. The accepted way of implementing this is to use monitors. An alternative method with slightly different consequences is an atomic multiple-store operation in the language. This avoids the necessity of ever having the object in a programmer-visible state that contradicts the invariant, at the cost of some programming awkwardness.

One place where this problem arises in particular is in the initialization of objects. Invariants cannot be guaranteed to hold until the object has been initialized properly. We first encountered this problem in the context of argument records: the function may have complex criteria for argument legality, which cannot be checked until all the arguments have been computed. In this case there is normally no problem, since the construction of the argument record and the storing of the argument values is an atomic operation from the programmer's viewpoint. The right way to take care of this seems to be the same as in Alphard, namely, to allow implementor-specified initialization code.

Topics not covered adequately

How do scopes get built out of dictionaries -- search rules.

Inter-Office Memorandum

To	EPE working group	Date	November 15, 1978
From	Lampson, Levin, Satterthwaite	Location	Palo Alto
Subject	Plan for an Interim Integrated Mesa Environment	Organization	CSL

XEROX

Filed on: [Ivy]<Levin>PE2>IME.bravo

This memo describes the steps required to obtain an interim integrated Mesa programming environment. We establish the following ground rules:

1) The intended user community for this environment is a small corps of experts who will use it to construct a more lasting integrated Mesa world. Consequently, the proposed system is not idiot-proof and occasionally delivers its function in less-than-ideal ways.

2) The primary purposes of this environment are to speed up the edit/compile/debug/edit loop by removing the Alto Executive as the link between weakly-communicating programs and to exploit the large real and virtual memories of the Dorado.

3) This system is only intended to work on the Dorado.

We divide the development into two phases. Phase I will be concerned exclusively with the construction of the environment proper and a fast edit/compile/debug/edit loop. A change to a module will require recompilation of that module and any other modules that depend upon it. Configurations whose constituent modules change will have to be rebound and reloaded, causing data in their frames to be lost. Phase II will produce a system that relaxes these requirements in certain common situations, thereby eliminating unnecessary recompilations and loss of data in these cases.

Approximate Effort to Construct Phase I System: 9.5 work-months
Approximate Effort to Construct Phase II System: 4 work-months

Phase I Development

Phase I represents the bulk of the effort. When completed, it will have produced an environment in which a text editor, the compiler, the binder, the debugger, and the user's program(s) are all simultaneously available. The components of the Phase I system are listed below with estimates of the effort required to produce them.

Microcode

The present Dorado Mesa microcode supports Mesa 4.1 and precisely emulates the Alto behavior. It is therefore extremely inconvenient to access more than 64K of primary memory. The microcode will have to be extended to be functionally compatible with the present D0 implementation, which requires the addition of instructions that manipulate long pointers (24 bit addresses) and page

faulting. *Approximate effort*: 1 work-month. (Note that this work will be done regardless of IME plans.)

System

The Alto/Mesa runtime environment will be carried over to the Dorado more-or-less intact. The actual movement of code requires only recompilation (to take advantage of the microcode discussed above). Functional changes are required to handle page faults and support (code) segment allocation in the 24-bit address space. Nearly all of this work has already been done for XMesa (the version of Alto/Mesa that exploits the extended memory of a wide-bodied Alto). The only additional substantive change is a virtual memory maintenance facility. We expect to implement a simple-minded scheme in which the virtual memory is mapped onto a contiguous region of the disk, thereby eliminating the need for paging tables. *Approximate effort*: 1 work-months.

An rudimentary protection scheme is required to keep the various programs from clobbering each other. We intend to give each major entity (compiler, debugger, editor, etc.) its own MDS (main data space - a 64K region of the virtual address space that is implicitly addressed by short (16-bit) pointers). Code segments and symbol tables (see below under Compiler) will reside outside all MDSs and thus be accessible only via long pointers. Since any program can fabricate a long pointer and use it, we cannot prevent arbitrary access, but by using the write-protect facility of the hardware memory map, we can prevent arbitrary modification. When a particular MDS is executing, all other MDSs, all code segments, and all symbol tables will be write-protected. Switching control from one MDS to another requires changes to the map entries for all pages in each MDS. Code segments and symbol tables remain write-protected except when the compiler and debugger find it necessary to modify them (e.g., setting breakpoints, cache update). *Approximate effort*: 2 work-months.

Interactive Base

We envision a window-oriented display manipulation facility as the basis for interaction with the various programs in the environment. The package presently used in the Mesa debugger supplies adequate function, albeit in a somewhat awkward fashion. It is likely that this interface will be replaced by the one used in the Tools Environment, possibly within the next few months. (This conversion would be done by the Mesa group and become part of the official, released debugger.) *Approximate effort*: 0 work-months (That is, we'll take what they give us.)

An alternative package (called Chameleon) is being developed by AOS to support market probes. This interface will probably share many of the ideas of the Tools interface, but package them somewhat differently. In particular, Chameleon expects to find a data base package (AOS calls it Iliad) and understands a variety of scenarios for interacting with the data base. Chameleon will supply a text editor as well (see below). *Approximate effort*: 0.5 work-months. (That is, we won't adopt this package unless it appears stable and we can use it with only minor tweaks.)

Text Editor

There are (at least) two sources for a text editor of adequate functionality (i.e., about what the Laurel editor provides). The Tools group have an editor that runs in the Tools environment. Obviously, if this environment becomes the interactive base of IME, we can adopt this editor virtually intact. (Smokey believes we can alter the command set easily to provide global substitution, etc., if we want.) The editor uses a quick-and-dirty form of piece tables (every piece is an integral number of lines), which means it is probably somewhat wasteful of disk space. *Approximate effort*: 1 work-month.

Chameleon will also provide a text editor, whose functions will resemble those of the Laurel editor (but cleaned up a bit). Doug Brotz is slightly skeptical about unplugging the Chameleon editor and inserting our own, but agrees that there should be no problem "in principle". The Chameleon editor will understand about data base records and manipulating fields of such records, a feature we might want to play with, but don't really need. *Approximate effort*: 1 work-month.

(The effort estimates above are intended to be upper bounds. Butler believes we can "hack together an editor in 2 work-months", so we probably shouldn't invest more than 1 work-month in some other package.)

Compiler

In Mesa, every module has an associated symbol table. Both the compiler and the debugger maintain a cache of the symbol tables they manipulate, though the limited memory space on the Alto tends to restrict the effective size of these caches. We propose to maintain this cache in the virtual memory space and change the compiler and debugger to share a single cache, whose entries are accessed by long pointers. Ed Satterthwaite believes this compiler change is nothing more than a recompilation with an altered definitions file. Once this separate cache exists, the compiler's "front-end" processing of the DIRECTORY statement (which maps symbol table names to Alto files) has to be reworked in a straightforward manner. *Approximate effort*: 2 work-months.

Debugger

In IME, the debugger will be "in control". That is, the command language (such as it is) will be interpreted in the debugger. Thus, the existing command processor will be extended to include commands that trigger editing, compilation, binding, etc. This will require that the Alto/Mesa debugger's interface to the "debugee" (currently, Inload/Outload) be replaced by the MDS-switching mechanism described above. Data in the debugee will then be accessed by long pointers instead of reading Swatee. The debugger will also be changed to use the common symbol table cache. *Approximate effort*: 1.5 work-month.

Binder

The Mesa binder combines configurations (of which the compiler output is a degenerate case) to form new configurations. In the Phase I system, when a configuration changes, all configurations in which it appears must be rebuilt. The binder will therefore be required as a part of the process that takes a multi-module program from editing to execution. We intend to make no functional changes to the binder. In particular, since the binder does not read symbol tables (except in a unimportant, Alto-specific case), it doesn't have to be modified to use the shared symbol table cache. *Approximate effort*: 0.5 work-months.

Phase II Development

Phase II will actually provide two extensions to the Phase I system. These extensions are independent and could be performed in either order.

Replacing an Existing Module

Small changes to a module should not necessarily require recompilation and rebinding of all configurations in the dependency chain. In particular, if the compiler is able to honor the existing structure of the module when recompiling it, the resulting new version can (almost) be slipped into the system as a replacement for the old version without altering the current state. Specifically, the compiler must retain the arrangement of variables in the global frame and the order of procedures

in the module's entry vector. (Additional entries can be tolerated as long as they don't require allocation of a larger global frame.) This ensures that links in other modules will not have to be changed. Obviously, the compiler cannot always satisfy these constraints, but when the module has been changed only slightly, the compiler has a good chance of doing so. *Approximate effort*: 2 work-months.

There are one complication in the loader, however. At the time that the (new version of the) module is loaded, the loader must locate all extant local frames that point to the global frame being replaced. The saved program counters in these frames are now nonsense, since the code has been recompiled. Therefore, these frames cannot be allowed to execute. Two options are available: (1) cut back the stack to the point of such a frame (i.e., simulate a RETURN from that frame), or (2) mark the frame so that a trap occurs when control returns to it. Both options are of equal difficulty. *Approximate effort*: 1 work-month.

Understanding Module Dependencies

SDD has, in the past, worked on a system called DeSoto, whose purpose is to automate the process of maintaining a consistently compiled and bound set of modules. DeSoto actually consists of three separable parts: (1) a dependency analyzer, (2) a symbol table patcher, and (3) a file system maintainer. The dependency analyzer is a straightforward program that follows the graph of module inclusions to determine what modules might (logically) require recompilation as a result of a change. The symbol table patcher has a detailed knowledge of symbol table structure and can determine whether a particular change actually requires recompilation or whether it can be accommodated by patching the symbol table (and associated .bcd). The file system maintainer makes sure that consistent source and object files are stored in designated places, and controls the other two parts of DeSoto.

We propose to implement only the dependency analyzer as part of the Phase II system. (In fact, the Mesa group has a stand-alone version of this program, called the Include checker.) The dependency analyzer by itself ensures that all modules that must be recompiled are in fact recompiled, though it will induce unnecessary recompilations. We consider the effort to implement the symbol table patcher to be of marginal benefit for IME. *Approximate effort*: 1 work-month.

Inter-Office Memorandum

To	EPE working group	Date	November 16, 1978
From	P. Deutsch	Location	Palo Alto
Subject	EPE consensus principles	Organization	CSL

XEROX

Filed on: <Deutsch>epe-consensus.memo on Maxc1

At the request of the working group, here is a memo which sets down certain "consensus principles" which are implicit in the EPE Report, although not explicitly stated, and which were made more explicit in the presentations that Butler and I gave at Dealer on Nov. 1.

Here are what both presentations mentioned as important strengths in Lisp (to be preserved in XLisp, and added to EPE Mesa):

Automatic storage deallocation

Easy use of programs as data

Atoms

Universal pointers / run-time type system

Integrated editor / instant turnaround

The remaining items on our lists did not coincide, although they all appear on the feature list in the EPE Report. My presentation mentioned Mesa strengths to be added to Lisp, but Butler's did not mention Mesa strengths to be preserved (presumably because EPE Mesa is meant to be upward compatible with present Mesa in a much more direct sense than XLisp is upward compatible with Interlisp).

Here are some mechanisms we both viewed as appropriate for representing information about programs:

Tables mapping name to <location, type> for frames

An atom table associating a print name (string) and internal ID

Tables for each pointer-containing type giving <location, type> for each pointer field

Globally unique type identifiers

"Spacing" tables mapping (fixed-size) ranges of virtual addresses to the type of objects allocated in that range

Inter-Office Memorandum

To	PE-Interest	Date	November 16, 1978
From	Lampson, Levin	Location	CSL, Palo Alto
Subject	Mesa-based EPE plan	File	[Ivy]<Lampson>EpeMesaPlan.memo

XEROX

The plan for a Mesa-based EPE has three major stages:

Preliminaries: an interim integrated Mesa environment (IME) which provides quick edit-compile-run turnaround for the existing Mesa language and system, as well as rudimentary access to the Dorado's large virtual memory. This system serves as a base for later developments.

Foundations: basic, low-level facilities needed to match the functionality of Lisp: atoms, variable syntax analysis, dynamic variable binding, a safe language subset, and automatic storage deallocation.

Features: equivalents for the Lisp Masterscope, file package, programmer's assistant, Helpsys and similar features, in a more general setting, together with improvements to the debugger and editor, facilities for specifying the construction of complex systems, and other "environmental" capabilities.

The stages must be done in approximately the indicated order, although some overlap is possible. An important property of the plan is that growth of the system is highly incremental: even within a stage, there are many points at which a stable system with improved function is available.

Preliminaries

This section describes the steps required to obtain an interim integrated Mesa programming environment (IME). We establish the following ground rules:

1) The intended user community for this environment is a small corps of experts who will use it to construct a more lasting integrated Mesa world. Consequently, the proposed system is not idiot-proof and occasionally delivers its function in less-than-ideal ways.

2) The primary purposes of this environment are to speed up the edit/compile/debug/edit loop by removing the Alto Executive as the link between weakly-communicating programs and to exploit the large real and virtual memories of the Dorado.

3) This system is only intended to work on the Dorado.

We divide the development into two phases. Phase I will be concerned exclusively with the construction of the environment proper and a fast edit/compile/debug/edit loop. A change to a module will require recompilation of that module and any other modules that depend upon it. Configurations whose constituent modules change will have to be rebound and reloaded, causing data in their frames to be lost. Phase II will produce a system that relaxes these requirements in certain common situations, thereby eliminating unnecessary recompilations and loss of data in these cases.

Approximate Effort to Construct Phase I System: 9.5 work-months
Approximate Effort to Construct Phase II System: 4 work-months

Phase I Development

Phase I represents the bulk of the effort. When completed, it will have produced an environment in which a text editor, the compiler, the binder, the debugger, and the user's program(s) are all simultaneously available. The components of the Phase I system are listed below with estimates of the effort required to produce them.

Microcode

The present Dorado Mesa microcode supports Mesa 4.1 and precisely emulates the Alto behavior. It is therefore extremely inconvenient to access more than 64K of primary memory. The microcode will have to be extended to be functionally compatible with the present D0 implementation, which requires the addition of instructions that manipulate long pointers (24 bit addresses) and page faulting. *Approximate effort*: 1 work-month. (Note that this work will be done regardless of IME plans.)

System

The Alto/Mesa runtime environment will be carried over to the Dorado more-or-less intact. The actual movement of code requires only recompilation (to take advantage of the microcode discussed above). Functional changes are required to handle page faults and support (code) segment allocation in the 24-bit address space. Nearly all of this work has already been done for XMesa (the version of Alto/Mesa that exploits the extended memory of a wide-bodied Alto). The only additional substantive change is a virtual memory maintenance facility. We expect to implement a simple-minded scheme in which the virtual memory is mapped onto a contiguous region of the disk, thereby eliminating the need for paging tables. *Approximate effort*: 1 work-months.

An rudimentary protection scheme is required to keep the various programs from clobbering each other. We intend to give each major entity (compiler, debugger, editor, etc.) its own MDS (main data space - a 64K region of the virtual address space that is implicitly addressed by short (16-bit) pointers). Code segments and symbol tables (see below under Compiler) will reside outside all MDSs and thus be accessible only via long pointers. Since any program can fabricate a long pointer and use it, we cannot prevent arbitrary access, but by using the write-protect facility of the hardware memory map, we can prevent arbitrary modification. When a particular MDS is executing, all other MDSs, all code segments, and all symbol tables will be write-protected. Switching control from one MDS to another requires changes to the map entries for all pages in each MDS. Code segments and symbol tables remain write-protected except when the compiler and debugger find it necessary to modify them (e.g., setting breakpoints, cache update). *Approximate effort*: 2 work-months.

Interactive Base

We envision a window-oriented display manipulation facility as the basis for interaction with the various programs in the environment. The package presently used in the Mesa debugger supplies adequate function, albeit in a somewhat awkward fashion. It is likely that this interface will be replaced by the one used in the Tools Environment, possibly within the next few months. (This conversion would be done by the Mesa group and become part of the official, released debugger.) *Approximate effort*: 0 work-months (That is, we'll take what they give us.)

An alternative package (called Chameleon) is being developed by AOS to support market probes. This interface will probably share many of the ideas of the Tools interface, but package them somewhat differently. In particular, Chameleon expects to find a data base package (AOS calls it Iliad) and understands a variety of scenarios for interacting with the data base. Chameleon will supply a text editor as well (see below). *Approximate effort*: 0.5 work-months. (That is, we won't adopt this package unless it appears stable and we can use it with only minor tweaks.)

Text Editor

There are (at least) two sources for a text editor of adequate functionality (i.e., about what the Laurel editor provides). The Tools group have an editor that runs in the Tools environment. Obviously, if this environment becomes the interactive base of IME, we can adopt this editor virtually intact. (Smokey believes we can alter the command set easily to provide global substitution, etc., if we want.) The editor uses a quick-and-dirty form of piece tables (every piece is an integral number of lines), which means it is probably somewhat wasteful of disk space. *Approximate effort*: 1 work-month.

Chameleon will also provide a text editor, whose functions will resemble those of the Laurel editor (but cleaned up a bit). Doug Brotz is slightly skeptical about unplugging the Chameleon editor and inserting our own, but agrees that there should be no problem "in principle". The Chameleon editor will understand about data base records and manipulating fields of such records, a feature we might want to play with, but don't really need. *Approximate effort*: 1 work-month.

(The effort estimates above are intended to be upper bounds. Butler believes we can "hack together an editor in 2 work-months", so we probably shouldn't invest more than 1 work-month in some other package.)

Compiler

In Mesa, every module has an associated symbol table. Both the compiler and the debugger maintain a cache of the symbol tables they manipulate, though the limited memory space on the Alto tends to restrict the effective size of these caches. We propose to maintain this cache in the virtual memory space and change the compiler and debugger to share a single cache, whose entries are accessed by long pointers. Ed Satterthwaite believes this compiler change is nothing more than a recompilation with an altered definitions file. Once this separate cache exists, the compiler's "front-end" processing of the DIRECTORY statement (which maps symbol table names to Alto files) has to be reworked in a straightforward manner. *Approximate effort*: 2 work-months.

Debugger

In IME, the debugger will be "in control". That is, the command language (such as it is) will be interpreted in the debugger. Thus, the existing command processor will be extended to include commands that trigger editing, compilation, binding, etc. This will require that the Alto/Mesa debugger's interface to the "debugee" (currently, Inload/Outload) be replaced by the MDS-switching mechanism described above. Data in the debugee will then be accessed by long pointers instead of reading Swatee. The debugger will also be changed to use the common symbol table cache. *Approximate effort*: 1.5 work-month.

Binder

The Mesa binder combines configurations (of which the compiler output is a degenerate case) to form new configurations. In the Phase I system, when a configuration changes, all configurations in which it appears must be rebuilt. The binder will therefore be required as a part of the process that takes a multi-module program from editing to execution. We intend to make no functional changes

to the binder. In particular, since the binder does not read symbol tables (except in a unimportant, Alto-specific case), it doesn't have to be modified to use the shared symbol table cache. *Approximate effort*: 0.5 work-months.

Phase II Development

Phase II will actually provide two extensions to the Phase I system. These extensions are independent and could be performed in either order.

Replacing an Existing Module

Small changes to a module should not necessarily require recompilation and rebinding of all configurations in the dependency chain. In particular, if the compiler is able to honor the existing structure of the module when recompiling it, the resulting new version can (almost) be slipped into the system as a replacement for the old version without altering the current state. Specifically, the compiler must retain the arrangement of variables in the global frame and the order of procedures in the module's entry vector. (Additional entries can be tolerated as long as they don't require allocation of a larger global frame.) This ensures that links in other modules will not have to be changed. Obviously, the compiler cannot always satisfy these constraints, but when the module has been changed only slightly, the compiler has a good chance of doing so. *Approximate effort*: 2 work-months.

There are one complication in the loader, however. At the time that the (new version of the) module is loaded, the loader must locate all extant local frames that point to the global frame being replaced. The saved program counters in these frames are now nonsense, since the code has been recompiled. Therefore, these frames cannot be allowed to execute. Two options are available: (1) cut back the stack to the point of such a frame (i.e., simulate a RETURN from that frame), or (2) mark the frame so that a trap occurs when control returns to it. Both options are of equal difficulty. *Approximate effort*: 1 work-month.

Understanding Module Dependencies

SDD has, in the past, worked on a system called DeSoto, whose purpose is to automate the process of maintaining a consistently compiled and bound set of modules. DeSoto actually consists of three separable parts: (1) a dependency analyzer, (2) a symbol table patcher, and (3) a file system maintainer. The dependency analyzer is a straightforward program that follows the graph of module inclusions to determine what modules might (logically) require recompilation as a result of a change. The symbol table patcher has a detailed knowledge of symbol table structure and can determine whether a particular change actually requires recompilation or whether it can be accommodated by patching the symbol table (and associated .bcd). The file system maintainer makes sure that consistent source and object files are stored in designated places, and controls the other two parts of DeSoto.

We propose to implement only the dependency analyzer as part of the Phase II system. (In fact, the Mesa group has a stand-alone version of this program, called the Include checker.) The dependency analyzer by itself ensures that all modules that must be recompiled are in fact recompiled, though it will induce unnecessary recompilations. We consider the effort to implement the symbol table patcher to be of marginal benefit for IME. *Approximate effort*: 1 work-month.

Foundations

We culled five major (read "potentially difficult to implement") facilities from the memos by Bobrow and Masinter, and discussions among the EPE group:

1) Programs as data
2) Variable syntax analysis
3) Dynamic variable binding
4) Safe language (subset)
5) Automatic storage allocation

These seem to be the major basic language/system capabilities which must be added to Mesa to make a Lisp-like programming style possible. Doubtless there are also a number of minor points which will need to be improved, but we believe that these are in the moise for the current level of planning detail.

Approximate total effort for Foundations stage: 20 work-months, including 4 work-months for cleanup of the current Mesa compiler. These estimates result from consultation with Ed Satterthwaite for compiler-related things, and with Peter Deutsch for things copied from Alto/Dorado Lisp.

1) Programs as data.

The ability to treat programs as data requires three major facilities within the Mesa environment: (a) an S-expression-like representation of source programs, (b) a means of using atoms (in the LISP sense) uniformly throughout the environment, and (c) a universal pointer mechanism.

S-expression representation.. This seems straightforward in principle, though we haven't looked at the details. We think that a representation other than parse trees should be explicitly defined, permitting the representation of parse trees to change as the compiler/language evolve.

Approximate effort: 1 work-month.

Atoms. The symbol tables built by the Mesa compiler currently include a mapping between source identifiers and internal unique ids (atoms). At present, these mappings exist only on a per-compilation unit basis, implying that they will need to be merged into a global map when the compilation unit "enters the environment". A similar merge presently occurs inside the compiler when the source program accesses an included module (i.e., the symbol table for that module is opened and the identifiers (used by the source program) are remapped into the current atom table). A merge into a global atom table therefore seems feasible, at least in principle.

Approximate effort: 1 work-month.

Universal Pointers. We will need pointers that carry the type of their referent with them. This in turn requires a scheme for universal type IDs, which do not currently exist in Mesa. It seems possible to extend the techniques used to ensure uniqueness of record types to accommodate general types. Run-time efficiency is vital, so the check that two type IDs are equivalent must be fast. We anticipate using the LISP "spacing" technique, which places all data structures of the same type within a well-defined region of the address space (e.g., each page allocated for such dynamic structures can hold objects of precisely one type). This scheme avoids keeping any type ID bits with the pointer, but makes it difficult to point inside a composite data structure with a universal pointer. Two possible remedies have been suggested: (1) an explicit type identifier immediately preceding the (sub)structure, or (2) an indirection mechanism in the data structure itself. In any event, access to statically allocated variables (i.e., those in local and global frames) via universal pointers will have to be prohibited, unless special steps are taken to prevent problems with garbage collection.

It will be possible to discriminate universal pointers with the existing Mesa facilities for discriminating variant records. We also intend to add a facility for coercing a universal pointer to a pointer of more specific type; the coercion must be acocmpanied by a run-time check. The intended model is the Lisp IPLUS function, which coerces its arguments to integers. This coercion is applied during assignment; since argument-passing in Mesa is treated exactly like assignment, this seems to work out just fine.

Approximate effort: 4 work-months.

2) Variable syntax analysis.

LISP provides three opportunities for the programmer to affect the parsing/interpretation of input text: (a) at scanning time, (b) at compile time, and (c) at function call time. (a) may be accomplished by providing scanner escapes to user-written code, which will replace some appropriate substring of the current parsing context (i.e., the as-yet unclosed lists) with a piece of program in S-expression form. It appears straightforward to provide a facility comparable in power to LISP's readtables -- a more elegant mechanism might be desirable. (b) may be accomplished by more-or-less conventional macro expansion along the lines of the Mesa mechanisms for in-line procedures. (c) is deemed undesirable and will not be provided.

Approximate effort: 4 work-months (3 for (a), assuming the existing Lisp read-table design, and 1 for (b)).

3) Dynamic variable binding.

Mesa, at present, can only support static binding of identifiers. To be able to bind a name to a variable in an enclosing *dynamic* context, we need to know what variables are to be found in each frame. Instead of the LISP default, we propose that variables to which dynamic binding may occur be declared as such at compile time. The compiler can then produce a table of entries of the form

atom -> <frame,type>

for each compilation unit. Such a table could reside in the code segment, with a bit in the associated global frame(s) to indicate its existence. Dynamic binding would then occur much in the same way the signal-to-catch phrase binding occurs (but more efficiently). It might be better to use some other form of variable identifier than its atom; the existing machinery used to bind signal codes should be suitable for binding such identifiers also.

Approximate effort: 1 work-month.

4) Safe language

By a safe language we mean here one in which a pointer value of a given type POINTER TO T is guaranteed to point to a region of storage containing a value of type T. We will also consider a relaxation of this condition in which certain *regions* of the storage may not be safe in this sense; as long as values stored in such unsafe regions do not themselves contain pointers to safe regions, the damage which can be caused is confined to the unsafe regions.

There are three ways in which a pointer variable *p* may acquire an improper value

1) Lack of initialization - in this case the value of *p* is just the random bit pattern which happens to be in the storage occupied by *p* when it is instantiated.

2) Incorrect computation - in general a pointer value can only be the value of some pointer variable (which could of course be a record component) or the result returned by a storage allocation. If pointer values can be computed in other ways, e.g. with LOOPHOLEs or their equivalents, or by arithmetic, then there is no guarantee that such values point to storage containing values of the proper type.

3) Premature deallocation - in this case *p* is pointing at a region of storage which once contained a value of type T, but is now being used for some other purpose.

If none of these bad things can happen, then induction tells us that the language is safe with respect to pointers. In addition, it is necessary to do bounds checking on array references, since otherwise every array index is as dangerous as a pointer. Finally, array descriptors require the same treatment as pointers.

Preventing uninitialized pointers is straightforward, by generating code to initialize all pointers in procedure frames when they are allocated, and in dynamic variables of record and array types when they are allocated. Mesa 5 will have language constructs for allocating dynamic variables, so it is known where to do the initialization.

Preventing incorrect computation requires excluding all the dangerous constructs from the language

a) Loopholes into a pointer type or a pointer-containing type (PC for short)

b) Computed or overlaid variants in which the variant part is PC

c) Pointer arithmetic

d) Use of UNSPECIFIED (this restriction is probably stronger than necessary)

e) What is missing?

Preventing premature deallocation has two aspects: dynamic (explicitly allocated) variables and stack variables. For the latter, a simple and reasonable rule seems to be to disallow the use of @ on local variables. As partial compensation, it seems desirable to introduce var parameters (call by reference); these can be thought of as pointers to stack variables which are *always* dereferenced, so that it is not possible for the called procedure to make a copy of the pointer which will outlive the call. We also need to enforce a restriction that var parameters are not permitted across process and coroutine calls, since in those cases there is no guarantee that the caller will live longer than the called procedure (again, this seems too strong, but it isn't clear how to weaken it).

For dynamic variables some support from an automatic deallocation mechanism seems essential, since otherwise there is no way to tell when the programmer says Free(*p*) whether other pointers to *p*↑ still exist. We see three possibilities

a) Reference counting. This could be added to Mesa easily enough, but the cost during execution is rather high, and it is probably not a good idea in view of the other alternatives.

b) Deferred freeing. The idea is to make a list of all the storage freed by the programmer, and at suitable intervals to invoke the scanning phase of a garbage collector and check that there are no outstanding references to the storage queued to be freed. At the end of the scan any storage which has passed this check can be actually freed.

c) Automatic deallocation, discussed below.

Quite possibly deferred freeing will be worthwhile, since it is fairly easy to implement and has substantially lower running costs than fully automatic deallocation. With plenty of address space there should be no serious objection to a long interval between the programmer's call of Free and actual availability of the storage.

Mesa currently has primitive and unsafe facilities for relative pointers, which there is a design to strengthen and improve. Relative pointers have several appealing properties, one of which is particularly relevant in this context

a) They allow an entire complex structure to be moved in storage, or between storage and a file, as long as all its contained pointers are relative to the structure.

b) They can save substantial amounts of space, especially as the address space grows and pointers become longer.

c) Misuse of a relative pointer can only damage data in the region it refers to. If none of the storage in this region contains any external pointers, then no global damage can occur. Embedded external pointers are still possible if they are treated as array indices, i.e. go through fingers. The advantage is that the programmer can forgo the safety precautions for such relation pointers without risking the integrity of the entire system. Of course a bounds check is still required (analogous to an array bounds check), but in many cases of interest it can be provided by storing the pointers in n-bit fields and making the age of the region 2^n; an especially nice value for n is 16.

Approximate effort: 2 work-months.

5) Automatic deallocation

Our proposal is to have automatically deallocated (AD) types, and to use the Deutsch-Bobrow scheme for incremental garbage collection. This requires knowledge of the location and type of every pointer to an AD type, but the techniques are well-known and straightforward.

Approximate effort: 3 work-months (for Lisp-style transaction garbage collection, including necessary compiler changes).

Features

xxx

XEROX
PALO ALTO RESEARCH CENTER
Computer Science Laboratory
December 8, 1978

To PE-Interest

From Dan Swinehart

Subject Mesa-Based EPE Programmer's Assistant

Filed as [Maxc and Ivy]<Swinehart>Assistant.bravo, . . .Assistant.Press

This section attempts to characterize the "programmer's assistant" facilities of a Mesa-based EPE, and to estimate their cost. Here we consider the activities required to provide a more or less direct replacement for existing Interlisp functions.

However, one must bear in mind that we have the opportunity to redesign the user's interface to the system, taking the high bandwidth display into account. The goal will be to provide a consistent, integrated default interface for program development and for control of user applications that do not have exotic interactive requirements. The right facilities for such an environment will look substantially different from previous ones, as environments like the Smalltalk browser, DLisp, and the Mesa Tools are beginning to show us. These issues deserve a great deal of additional design work, once the project is underway.

Prerequisites

It is assumed that the work described under *Preliminaries* and under *Fundamentals* has been completed. The text editor and debugger described in those sections, perhaps augmented by Mesa Tools, would certainly provide an adequate interface. The time estimates are presented in the same spirit of optimism that pervades the rest of this document.

Interpreter

Before anything remotely resembling the Interlisp interactive flavor can be achieved, either the Mesa debugger's interpreter must be extended to encompass the entire language, or a compiler-based statement evaluator must be produced. In this latter, preferred, approach, the compiler would be extended to provide the equivalent of incremental compilation for single non-declarative statements in specified contexts. An evaluator within the debugger would generate the appropriate linkages to get the statements executed. Sufficient information would be saved to provide the History List features (below.) An appropriate default global context could be made available, if necessary, for the execution of statements corresponding to the Lisp "top level". Suggested implementation: enclose the statements in a procedure body, compile and execute the procedure.

Approximate effort: interpreter, 4-6 somewhat mythical work months; compiler extension, 2-4. Evaluator, top-level environment, .5-1.0 work months. If all or part of this work is implicit in providing the "programs as S-expressions" fundamental capability, the estimates can be reduced accordingly.

Break and Advise

Only modest extensions to the current conditional breakpoint code, using the incremental compiler invented just above, are necessary to obtain a break/advise facility the equal of Interlisp's -- more powerful, in fact, when combined with the Mesa display-oriented methods for indicating points of interest. (In fairness, DLisp presumably restores the balance.) The standard breakpoint handler could take care

of providing the linkage to the system-produced procedures containing the advice.

Approximate effort: 1-3 work months.

History Lists and their Use

The top-level interpreter, a history list that records top-level user actions at the text level, along with an indication of the contexts in which they were executed, are sufficient to implement the *Redo* command and the function-composition capabilities that it enables. But to provide *Undo* requires that internal information be used as well. Once automatically allocated lists, programs as S-expressions, atoms and data with attached types, and the other *fundamentals* are available, implementation of both commands should be straightforward. Probably both would be done, as in Lisp, by retaining S-expression representations of the original statements and their reversing functions. The following points prevent a placement in the easy category:

> In Mesa even simple programs consist of several independent global contexts. Virtually every user action will have to be interpreted with respect to a specific environment. History list entries will have to include the corresponding environmental information, and the commands will have to know when the corresponding environments no longer exist, or will have to arrange, as Interlisp can, to retain the context frames as long as they are referenced (requires garbage collector to operate on frames.) Contexts running as concurrent processes could also cause trouble if more than one of the processes becomes involved in user history activity. The history provisions would want to be more fully integrated with the complex context structures than are the Interlisp models.
>
> Mesa programs, because Mesa is an assignment-based language, produce more temporary side-effects, on average, than Lisp programs. The information stored to provide undo capabilities might be more extensive. Because of the large virtual memory, space problems will probably not arise, but performance problems might. However, an implementation the equivalent of the TESTMODE(T) facility in Interlisp -- saving the old contents for *every* store into memory -- could cause both time *and* space problems.
>
> Mesa, possessing a syntax, is susceptible to syntax abuse. Such could occur when trying to provide the undoable versions of each language primitive that can produce a storage modification (e.g., a /← b ?) Some care will be required to integrate this basic capability into the language. This is just an instance of the efforts that are needed anyway to eliminate the special role of primitive generic functions in Mesa. Languages like Alphard provide a model for how this might be done.
>
> (More generally, as mentioned elsewhere in this report, the maintenance of a syntax that is acceptable to both man and LALR machine will present problems throughout the system that are not encountered in the pure Lisp implementation. It will be extremely difficult to produce a set of interactive facilities in Mesa whose use is as natural and compelling as those in the current Interlisp.)

Approximate effort: 2-4 work months to implement the mechanism; development of primitives and system procedures with undoable versions as time goes on.

Editor

If the editor chosen for the *Preliminaries* implements an undo/redo capability, integration with the more general version should be straightforward. Otherwise, these commands would have to be built from scratch, at considerable cost (depending on the editor design.)

Approximate effort: 1 work month or ? work months.

In a display-based editor, the structure-specific stuff can be largely implemented as part of the selection

mechanism. To the extent that the text files are in communion with the underlying parse tree or S-expression representations, it will be easy to produce the needed hierarchical selections.

Approximate effort: 1 work month.

Dwim

Some spelling correction and the repair of minor typographically induced syntax errors could be incorporated into the compiler. It would be useful, given room, in both interactive and standard versions of Mesa. The user interface to these tools are likely to be quite different in a system based on compilation and display editing. No estimate.

MasterScope

The structures needed for the *Fundamentals* implementations are adequate for the development of a Masterscope-style database. An extension of the interface used for the other tools would provide user access. The compiler, binder, or an intermediate agent, would notice change and update the data base.

Approximate Effort: the database facilities, no estimate; the interface, 2 work months.

Inter-Office Memorandum

To	Distribution	Date	December 13, 1978
From	Second EPE working group (names on next page)	Location	Palo Alto
Subject	Report from second Programming Environment Working Group	Organization	CSL

XEROX

Filed on: Ivy <Deutsch>pe2>
pe2.press - the report proper
{memos.press - technical memos submitted to the working group (the appendix to the report)
{minutes.press - the minutes of the working group's meetings

Distribution:
CSL
SSL: LRG, A. Henderson
SDD Palo Alto: Irby, Liddle, Lynch, Metcalfe, Satterthwaite, Dave Smith, Wallace, Weaver, White, Wick
ASD Palo Alto: Brotz, Elkind, Spitzen, Wegbreit

Non-CSL recipients may distribute further copies as needed

Table of Contents

1. Summary
2. Introduction, charter and history
3. Technical issues
    a. Global issues
    b. Facilities to be provided
    c. Character of resulting system
4. Organizational and resource issues
    a. Effort and schedule
    b. Participating people
    c. Project organization
5. Foreign affairs
    a. Importation
    b. Exportation
    c. Image

Appendix - technical memos submitted to the working group

Summary

The first CSL Programming Environment working group met intensively for a month last summer, and concluded that CSL should launch a project on the scale of the Dorado to develop a programming environment for the entire laboratory, starting from either Lisp *or* Mesa. A second working group has met weekly for nearly three months, exploring in more detail the probable consequences of the alternative starting points. This report presents those consequences in what we hope is sufficient detail for CSL to choose a starting point, and begin the actual construction of an Experimental Programming Environment.

Although the members of the working group are not in complete agreement about the choice of a starting point, there is almost no disagreement about technical issues or long-term goals. The facts and conclusions presented in this report are not in dispute; we have attempted to identify the (few) sources of unresolved disagreements.

As later sections of this report will make clear, the choice between Lisp and Mesa cannot be made solely on the basis of technical considerations. To within the uncertainty of estimation, both starting points present challenges of comparable difficulty, and would lead to systems of comparable utility, with comparable investments, over comparable time spans (two to three years). We are agreed that it is necessary and feasible for an EPE based on either system to support comfortably the programming styles currently associated with Lisp and Mesa. Some differences would remain in the character of the result; the value judgements placed on these differences generated most of the heat in our discussions. Social and political factors -- including the availability of people to fill key roles in the project, and our relations with the rest of the corporation -- are secondary (but possibly decisive) concerns.

Technical issues: Our further investigation of both the Lisp and Mesa alternatives led to two plausible development plans for producing a rudimentary environment on the Dorado (primarily for use by the EPE developers) in about six months, a widely-usable environment in a year and a half, and one superior to either of our present environments in about three years. The final system would provide all the facilities that present Lisp and Mesa users value highly: from Lisp, tools similar to the existing array of tools in the current Interlisp, and the ability to support integrated sublanguages and to delay bindings; from Mesa, the provisions for modularization with explicit interfaces, and the amenability to static checking. However, the system would probably retain some of the "flavor" of its starting point.

Organizational issues: Although the efforts required are about the same, a somewhat larger number of qualified people are available to work on a Mesa-based EPE. This report also discusses a possible structure for the project; after the language choice is made it should be possible to develop an internal operating plan quickly.

External affairs: The rest of Xerox has a fairly large and growing commitment to Mesa, and none to Lisp. Remaining largely compatible with the rest of the corporation has both advantages and disadvantages, but the advantages predominate. Either choice will reduce communication with important (but different) research communities in the outside world: Lisp favors the AI community, Mesa favors the programming language community.

Your working group respectfully asks CSL as a whole to:

endorse the immediate establishment of a large-scale EPE project;

based on the information contained in this report, and such other information as it considers relevant, choose the starting point for the project; and

agree to relegate further pro-and-con language discussions to the status of "printing discussions" for at least a year.

Charter and History

The purpose of this report is to set forth the discussions and conclusions of the second CSL Programming Environment working group. The most active members of this group (those who attended meetings regularly and produced memos or other substantial input for the group's deliberations) were:

Dan Bobrow
Peter Deutsch
Jim Horning
Butler Lampson
Roy Levin
Larry Masinter
Gene McDaniel
Jim Mitchell
Ed Satterthwaite (SDD)
Nori Suzuki
Dan Swinehart
Warren Teitelman

Many other members of CSL also attended some or most of the meetings and/or provided written material for the group.

We began with the conclusions of the first working group, reproduced here from their report:

> We think that CSL can support only one major PE, and that it can be evolved from either Lisp or Mesa; Smalltalk is farther from our goals, though certainly not out of sight. If a new PE is to be developed from Lisp or Mesa, then other work on the existing systems must be kept to a minimum during this development (unless we want half the lab's effort to be devoted to PEs), and they must be phased out as the new environment becomes viable. There doesn't seem to be any reason to attempt a multi-language PE, since an environment with all our ABC features would support any existing style of programming, while a two-language system would require more work, reduce synergy, and create artificial barriers. Our problem, then, is to decide whether we want to pay the price for a new programming environment:
>
> > agreement on what the starting point should be;
> >
> > manpower to build the ABC features which are missing;
> >
> > willingness of users to suffer substantial inconvenience in changing over.
>
> We are clearly talking about a project on the scale of the Dorado, and we should give it just as much careful thought as we did the Dorado project. Lab-wide discussion, and written feedback, seem like appropriate ways to carry out this process.

Our discussions focused on a number of areas in which we felt clearer understanding was needed before we could make a proposal to the lab as a whole about what to do in the PE area:

> What are the key technical issues which arise from each of the possible starting points?
>
> What differences in the ultimate system would necessarily follow from each choice of starting point?
>
> How much effort would be required to reach various levels of result from each starting point? What people would be available to do the work?
>
> What other non-technical and external considerations should play a significant role in our decision about what to do?

The group has been meeting once a week for approximately 3 months. We did not begin with a fixed idea of how long it would take us to reach satisfactory answers to the questions above. However, we feel that we now have sufficiently good answers to consider this phase of EPE activity finished: the next step is to make a final decision about the starting point, which will require discussion by the lab as a whole, and then get down to work on the EPE itself.

Introduction

Why does CSL need a new Experimental Programming Environment?

CSL's impressive productivity over the last few years can largely be attributed to its continuing investment in tools for itself. Dorado is a good recent example. It provides new opportunities for research in many areas by removing hardware capability (speed, memory, address space) as the major bottleneck. For the next few years, our ability to conduct computer system experiments will be largely limited by our programming capabilities.

CSL currently has two major programming environments, Interlisp on Maxc and Mesa/Bravo on the Alto. Each of these has major limitations deriving from its history and hardware base that make it less than ideal for at least some CSL applications. The arrival of Dorados provides an opportunity to remove many of these limitations and provide a *single* programming environment for the entire laboratory.

We have focussed our attention on the foreseeable needs within CSL in the next few years, with particular attention to "experimental programming." We have taken experimental programming to mean the production of moderate-sized systems that are usable by moderate numbers of people in order to test ideas about such systems; Bravo, Laurel, and KRL (the implementation part) were taken as typical of such experiments, but we believe that it will be important to conduct future experiments more quickly and at lower cost.

A programming environment is more than just a programming language. It is the entire system that is used by the programmer in the process of developing programs. As such, it will certainly contain such tools as text editors, debuggers, prettyprinters, etc. The underlying design principle is that the purpose of a programming environment is to facilitate programming:

> A good programming environment will reduce the *cost* of solving a problem by software. The cost will include the time of programmers and others in design, coding, testing, debugging, system integration, documentation, etc., as well as of any mechanical support (computer time, etc.). Since our human costs continue to be dominant, one of the most convincing arguments in favor of a particular programming environment feature is that it speeds up some time-consuming task or reduces the need for that task (e.g., it might either speed up debugging or reduce the amount of debugging needed). Software bottleneck removal is the implicit justification for many features.
>
> A good programming environment will also improve the *quality* of solutions to problems. Measures of quality include the time and space efficiency of the programs, as well as their usability, reliability, maintainability, and generality. Arguments relevant to this category tend to be of the form "this feature reduces the severity of a known source of problems," or "this feature makes it easier to improve an important aspect of programs." Thus a feature might be good because it reduces the frequency of crashes in programs or because it makes it convenient to optimize programs' performance.

We don't know how to quantify quality, but we did think about how much more productivity we might expect for sizable projects (e.g., Laurel, Bravo or the travel planner), as compared to the current state of affairs in either Lisp or Mesa. We guess that a factor of four is possible, about half from relaxing current space and time constraints by moving to the Dorado, and half from a PE which has the high-priority features on which we have agreed.

How will all this productivity be applied? We anticipate three major effects.

First, many more interesting things will be within the scope of a single person's efforts. Hence, the number of Laurel-sized projects can be expected to increase dramatically; not only are they much less work to do, but it is much easier to organize a one-person that a four-person project.

Second, much more elaborate things will be feasible, and hence will be attempted.

Third, the evolution of good packages that can be easily used without disastrous time, space or naming conflicts will cause a qualitative change in the nature of system-building: the current Interlisp system gives us our few hints of what this change will be like.

Global technical issues

Consensus principles

In the course of our discussions, a few principles emerged which were not captured as "features" in the original report, or were so fundamental that we thought they needed to be singled out for special mention. These principles included:

> Automatic storage deallocation is an absolute necessity. It produces a qualitative improvement in the ease of programming and the reliability of the results. This need not preclude another class of variables with programmer-managed allocation, but the latter must not be able to destroy the data structures necessary for the former.
>
> Easy use of programs as data underlies many other facilities in the system. Implementing this seems to require having Lisp-style atoms, a run-time type system, and universal pointers (pointers that carry a type with them).
>
> Editing facilities closely coupled with the compiler and the executive interface to the system are essential, both to reduce the turnaround time for minor changes, and to allow easy construction of tools that interact with these facilities.
>
> Capabilities for precisely defining interfaces, and restricting the communication between modules to those interfaces, are essential to reliable and readable programming. This includes, as a special case, the ability to restrict the use of types and names to a local lexical context.
>
> The ability to perform static checking (including, but not necessarily limited to, type checking) over designated program regions is necessary for both security and efficiency reasons.
>
> The present Lisp, Mesa, and Smalltalk programming styles must *all* be supported in a satisfactory way. The same packages, and tools of equivalent power, must be available in all styles. In particular, the Lisp capabilities for embedded variant languages and for programming entirely without type declarations must be supported.
>
> Both large, multi-person and small, single-person styles must be supported well. Bringing a wider range of experiments within the scope of a single person's effort is an important EPE goal.
>
> The EPE must support a wide range of binding times, including the Mesa and Smalltalk extremes, in a way that allows changes in binding time without structural changes in the program. Different choices of binding time by the programmer may lead to different turnaround times for apparently minor changes, and to different execution efficiencies, but the *functional* behavior of programs must not depend on such choices.

Smalltalk

Fairly early in our discussion we decided that either Lisp or Mesa would be preferable to Smalltalk as a candidate for the EPE base system -- sufficiently preferable that we did not need to consider Smalltalk further. This decision was based on the following observations:

> The present Smalltalk implementation is written in Alto assembly language and intrinsically cannot support more than 128K of real memory or 64K addressable objects. A major redesign and reimplementation of the lowest level of the system would be required to make it suitable as an EPE base. Neither Lisp nor Mesa has this problem.
>
> There is no evidence that large systems can be written in Smalltalk; the Smalltalk view of the world is enough different from that of either Lisp or Mesa that fundamental problems

may be revealed if we try to apply it to our larger projects. Large systems have been successfully written in both Lisp and Mesa.

Significant groups in CSL are familiar with either Lisp or Mesa; there is no CSL user community familiar with the Smalltalk view of the world.

The present direction of Smalltalk evolution is towards smaller machines (NoteTaker), while EPE should take significant advantage of the Dorado's capabilities.

However, there are a number of ideas in Smalltalk, not present in either Lisp or Mesa, which we believe are important to provide in the EPE:

The notion of class-structured behavior, and the ability of more specialized classes to inherit behavior from more general classes.

The user interface style.

The lack of structural distinction between user- and system-defined objects.

The working group spent essentially no time discussing these questions: it is up to the eventual project participants to make sure they receive proper attention.

(EP)E vs. E(PE)

At the beginning of our discussions, we assumed that the facilities required for E(PE), i.e. work on programming environments per se, were a subset of those required for (EP)E, i.e. construction of experimental systems in general. Consequently, we did not discuss the E(PE) parsing separately. However, it should be noted that the plan for a Lisp-based environment was strongly influenced by a desire to facilitate further work in language and system construction, since in the Lisp programming style, every medium- to large-scale project develops some language and system extensions; such considerations were much weaker in the Mesa-based plan. Several members of the working group with strong interests in programming languages and systems feel that neither proposed plan is likely to lead to a system that supports their interests for at least a couple of years.

Facilities

Our discussion of technical issues centered around the priority A, B, and C items from the first EPE report, reproduced below. The columns in the list have the following meaning:

Serial number - a serial numbering of the items, for later reference in this report. These are numbered in priority order, not in the L-P-T-X numbering system of the earlier report.

F/I/A code - the first working group's opinion of how important it was to provide for this item from the beginning in the system design.

F - Fundamental, much harder if not allowed for originally

I - Intermediate, somewhat harder if not allowed for

A - Add-on, difficulty does not depend significantly on pre-planning (although it may be intrinsically hard anyway)

Amount of discussion - the amount of time we spent discussing the item in the current working group.

0 - didn't even mention it
1 - mentioned in passing
2 - significant discussion
3 - major issue

L,M,S difficulty codes - the first working group's conclusion about how difficult it would be to implement this item in Lisp, Mesa, or Smalltalk respectively.

0 - available
1 - easy
2 - straightforward but takes time
3 - hard

Serial number	F/I/A code	Amount of discussion	L,M,S difficulty codes	Name of item
Priority A				
(A1)	F	3	030	Object management -- garbage collection/reference counting
(A2)	F	3	203	Statically checked type system
(A3)	I	0	000	Memory management -- object/page swapping
(A4)	F	3	302	Abstraction mechanisms; explicit notion of "interface"
(A5)	I	3	020	Fast turnaround for minor program changes (<5 sec)
(A6)	F	3	?0?	Adequate runtime efficiency
(A7)	F	1	002	Large virtual address space (≥ 24 bits)
Priority B				
(B1)	F	3	302	Encapsulation/protection mechanisms (scopes, classes, import/export rules)

(B2)	F	1	333	Well-integrated access to large, robust data bases
(B3)	I	3	020	Self-typing data (a la Lisp and Smalltalk), run-time type system
(B4)	A	2	203	Consistent compilation
(B5)	I	1	323	Version control
(B6)	F	2	000	Source-language debugger
(B7)	A	1	121	Text objects and images
(B8)	I	1	021	Uniform screen management
(B9)	A	3	202	User access to the machine's capability for packed data (see A6)
(B10)	F	3	111	Run-time availability of all information derivable from source program (e.g. names, types, scopes)
(B11)	F	2	222	CSL control over the system's future

Priority C

(C1)	A	0	102	Direct addressing for files (segmenting)
(C2)	I	0	000	Some support for interrupts
(C3)	I	3	021	Compiler/interpreter available with low overhead at run time
(C4)	I	0	012	Adequate reference documentation
(C5)	A	1	222	Librarian, program-oriented filing system (incl. Browser)
(C6)	I	3	012	Program-manipulable representation of programs
(C7)	I	1	011	Dynamic measurement facilities
(C8)	A	0	222	Scanned (bitmap) objects and images
(C9)	A	0	120	Press files
(C10)	F	0	010?	"Efficient" interface for experts
(C11)	A	0	222	Line objects and images
(C12)	A	1	001	Remote file storage

The two following subsections of this report cover all the topics identified as "major issues" in the above list, as appropriate for the individual language.

Mesa facilities needed in Lisp

The following facilities from the ABC list are present in Mesa but not adequately available in Lisp:

(A2) Statically checked type system
(A4) Abstraction mechanisms, explicit notion of "interface"
(A6) Adequate run-time efficiency
(B1) Encapsulation/protection mechanisms
(B4) Consistent compilation
(B9) (in some contexts) User ability to pack data

In the Lisp tradition, we propose to add these facilities by defining new objects to represent types, name scopes, etc. and associated functions for manipulating them. These objects are accessible to the programmer at run time; however, greater efficiency can be obtained by compiling programs with respect to a particular state of these objects, and paying the price of recompilation (or, more typically, reversion to interpreted or less fully compiled status) if a change is made. A similar comment applies to static checking -- a change in a type or scope requires re-checking the affected parts of the program.

The remainder of this section describes the principal features of a Lisp dialect we have been calling XLisp, which we propose as the EPE base. XLisp includes the current Interlisp facilities as a special case, but it is based primarily on SCHEME, a lexically-scoped Lisp dialect developed at MIT, with additions motivated by the Mesa type and scope system. We intend XLisp not only as a response to the EPE challenge, but as a conceptual framework for thinking about scope and type issues which has some room for future exploration of capabilities not present in either Lisp or Mesa.

To summarize how we propose to provide each of the six facilities listed above:

Static type checking -- by making types be objects, and (as needed) checking that all users of a declared variable refer to identical (EQ) type objects.

Explicit interfaces -- by defining an object called a dictionary that generalizes the notions of parameter list, record, and interface, and by requiring that such an object be associated with every defined or imported function.

Greater run-time efficiency, packed data -- by making the internal representation of a quantity be one of its attributes in the type system, and extending the instruction set to allow more efficient execution when more tightly bound representations are used.

Encapsulation/protection -- by associating protection information with entries in dictionaries.

Consistent compilation -- by extending the notion of type identity and compatibility into the permanent filing system, through an object called a permanent pointer which somewhat resembles Mesa's time stamps.

Dictionaries

A new kind of object called a dictionary is central to the interpretation of names (and, hence, programs) in XLisp. A dictionary consists of one or more name tables, each of which applies to a different syntactic context ("function" or "variable") and specifies the interpretation of a set of names in that context. Each entry in a name table consists of a name (atom) and an associated property list. Dictionaries implement function argument and result lists, record types, and interfaces: the latter uses are discussed more fully under Types below.

Dictionaries are the building blocks for scopes. Conceptually, a scope is an area of a program within which the interpretation of names does not change; implementationally, it is an object which specifies how to determine the meanings of names. (We will freely confuse the concept and

implementation in the discussion that follows.) A scope is made up of components, each of which gives the meaning of some names, and a rule for deciding how to resolve conflicts and what to do with undefined names. Each component may be a scope itself, or it may be a dictionary, which actually associates names and meanings.

Every name in a program must eventually resolve to a meaning of one of three basic kinds: manifest, in which the actual value is found in a dictionary (for example, local and system functions, record names, and what are now thought of as compile-time constants); dynamic, in which the value is found in a run-time data structure (for example, most variables and non-local functions); or unbound, in which only declarative information (if that) is available. In principle, every dynamic name eventually reduces through macro-expansion to a value obtained by applying elementary operations (at roughly the instruction set level) to constants and to pointers which form part of the virtual machine's state, such as the current stack frame pointer. These elementary operations are available as functions in the system, since they are needed for the interpreter and compiler, but their use is deliberately rendered inconvenient since it can lead to violations of scope rules on which not only compiler optimizations but the correct working of programs may depend. For ordinary use, we provide, instead of these primitives, a set of higher-level constructs for use in name tables: these constructs lend themselves better to static analysis and permit enforceable protection boundaries. These constructs are the following:

(MANIFEST value) -- the actual value is present in the name table. The value may be any Lisp object (data structure, function object, etc.). The value is treated as a constant: changing it may require recompiling any function that uses it. If it is a non-scalar value, it is only given out in read-only form.

(SCRATCH value) -- the value is present in the name table, but is not read-only. One can think of this as an OWN structure in the Algol sense.

(MACRO function-object) -- the name table contains a function which is applied to the form (application or atom), and the result is taken in place of the original form. It is vital that macros produce consistent results, i.e. only depend on the form being expanded and the scope in which it is being expanded. We assert that is sufficient to analyze the macro function and all functions it calls to make sure that they refer only to local and manifest variables, and to pass the macro function its arguments in read-only form; then any operations with side effects, if applied to objects not created during this call of the macro function, will cause an error.

(FLUID fluid-cell) -- this is the traditional Lisp binding method. Note that function and variable names are different for fluid binding purposes, and that the name itself is not a global atom but an object local to the dictionary, so that the binder and the user of a fluid name must share the dictionary.

(UNBOUND)

(LOCAL location-in-frame) -- this is the SCHEME lexically bound variable. It differs from a fluid variable in that closures which use local variables free retain a shared pointer to the binding in effect at the time the closure is *created*, while a free use of a fluid variable refers to the binding at the time the closure is *applied*. LOCAL names are also used for fields in records: in this case, location-in-frame defines the location of the field within the record instance.

(RELATIVE name scope base) -- this is used to refer to objects found in other scopes. Examples include argument or outer PROG variables referenced from within a PROG, instance variables and abstract operations in a record, and individual entries (usually manifest or fluid) imported from other scopes. A relative entry contains three parts: a name, a pointer to the scope in which to interpret the name, and, if the interpretation is local, an expression (interpreted in the original scope) which gives the base pointer. This base pointer becomes the default base pointer for the purpose of the interpretation, so

cascading of relative entries will work properly. The Mesa OPEN construct amounts to making appropriate RELATIVE entries in the current scope for all the names in the object being opened.

Types

Framework

Types are objects, just like integers or lists. There are three kinds of types: interface or I-types, which describe how objects behave in general; descriptive or D-types, which specify predicates that hold true of objects in particular situations; and representation or R-types, which describe the bit patterns used to represent objects inside the machine. Conceptually every object carries its I-type with it at execution time, but not its R-type (D-types are associated with variables, not with objects). The system provides a default representation for every I-type. The phrases "X is of type T" and "X satisfies type T" are interchangeable -- I- and D-types are predicates.

I-types are built up from elementary types like INTEGER and ATOM by (possibly recursive) applications of the following operators (the syntax is for expository purposes only):

SATISFIES[p:FUNCTION[[t:Type],[Boolean]]] is a type for objects x of type t such that p[x] is true.

ANYOF[t1 ... tn:Type] is a type for objects which can be of any of the types t1 ... tn.

ALLOF[t1 ... tn:Type] is a type for objects which must be of types t1 ... tn simultaneously.

READONLY[t:Type] is a type for objects which are of type t shorn of any operations which produce externally visible side-effects (more on side-effects below).

UNIQUE[t:Type] is a type which is the same as t, except that objects of this type are distinguishable from objects of type t and of any other type created by UNIQUE[t]. This operation is sometimes called "painting".

FUNCTION[argT,resultT:Type] is a type for functions which take arguments of type argT and return results of type resultT. ArgT and resultT must be structure types.

VARID[t:Type] is a type for variable id's of type t. Variable id's are used to link together binders and users of variables, leaving the identity of the variable unbound. (Same idea as the Mesa SIGNAL linking mechanism.)

Structuring -- read on.

The other kind of definable I-type is a structure type. It has named components, each of which may have type declarations. There are two kinds of components: fields, which obey certain rules set forth below that make them act like variables, and operations.

Every field component has two associated operations called fetching and replacing that field. These operations obey the following rules:

(1) Fetching a field has no externally visible side-effects (although it may have internal side-effects like caching or statistic-taking -- obviously "visible" can't be defined rigorously). In particular, it does not affect the contents of any field.

(2) Replacing a field has no externally visible side-effects on any operation other than fetching that same field.

(3) Fetching a field always returns what was last replaced into that field.

(4) No operation other than replacing a field affects what fetching that field will return.

The default R-type for fields is a storage cell: if the programmer specifies a different R-type, its adherence to the above rules is up to him.

Fields in structure types may have names, or an ordering, or both. The ordering is for argument and constructor lists -- it has nothing to do with storage format.

Arrays are a structure type in which the fields, instead of having names, are indexed by a set which may be of any type. If the type is an integer range, the default representation is a traditional array; if not, the default is a hash array.

An interface type, in the Mesa sense, is a structure type with fields which are of various function types. A Mesa module is a group of functions which all agree to use the same instance(s) of the interface types they import. Making an instance of a module consists of making an instance of the interface type it implements, filling in the fields with closures of the exported functions in which the imported interface pointers get bound.

Declarations

A D-type declaration just establishes an invariant for a variable -- checked whenever the variable is stored into, can be assumed whenever the variable is used. D-types can also be used as predicates, and wrapped around expressions. R-type declarations specify the representation of a quantity and hence are only meaningful for variables, as opposed to arbitrary expressions. ("Variables" include record fields, of course.)

Conformity and coercions

Type checking depends only on the I- and D-types of the objects. The conformity relation is straightforward to compute, given the type composition primitives. The compiler does all it can, and then produces code to do the remainder of the check.

Coercions are meaningful at both the I-type and R-type levels. I-type coercions up or down the type lattice are mostly either no-ops or type checks, and can be supplied automatically. Other I-type coercions require user-supplied functions. R-type conversions require user-supplied functions to convert to and from the default R-type, the familiar Lisp typed pointer.

Protection

We implement compile time checking by providing information in a dictionary that specifies the external accessibility on a name-by-name basis. The actual access rights are determined by a function of the originator, the path used to arrive at the dictionary entry, and the information in the entry itself.

Mesa provides four kinds of protection that are not checkable at compile time:

1) Variables, unless otherwise declared, are not accessible at run time by programs outside the lexical scope where they are defined.

2) A client and an implementor must use identical (EQ) versions of an interface.

3) An implementor can define types whose contents are (partially) private, but which the client can use to declare variables.

4) The types of the arguments passed in a control transfer always agree, regardless of how the environment receiving control was obtained (e.g. by binding, through a PORT, through a procedure variable).

In Mesa, all of these are based on intra-module checking by the compiler, and inter-module enforcement based on (2). This, in turn, is accomplished by creating a unique name for an interface when it is compiled, and copying that unique name into both the client and the implementor when they are compiled.

In XLisp, the privacy information in dictionaries and scopes provides (1), since the only way to access a non-local variable is by going from a frame or other object to the dictionary that describes it, and the operations to access objects in general are implemented within dictionaries and check the privacy information.

Our type checker provides (2) by simply using EQ -- XLisp always keeps with any compiled object a pointer to the dictionaries accessed during compilation, which include all imported and exported interfaces. The permanent pointer mechanism described under Miscellaneous below guarantees that the EQ relation is correct even when externally filed data are involved.

(4) is also a type checking question. If the user declares procedure and environment variables appropriately (which is automatic if an interface dictionary is involved), then the check can be made when the interface is bound rather than at the time of the control transfer.

The heart of our protection system is (3). Something like (3) is required to be able to have virtual objects, and objects accessible only through interface procedures (operations). In XLisp one can declare a record type as having scalable instances. What this means is that the record has a piece of code associated with it, and a "sealed" flag. If the record is in unsealed state, anyone may access it freely. If the record is in sealed state, only the associated code may access it. A record must be sealed to be used as the local environment of a function: the operation of calling a function logically breaks down into the steps of creating an instance of its local frame type, filling the parameters into the instance, sealing the frame, and then running the function. Similarly, records which define Smalltalk-like instances are sealed in a way that accepts incoming messages and then executes the implementing function. Such functions can access the sealed record by having a link in their scope structure that says they implement an operation on that record. (This is no less safe than the IMPLEMENTING construct in Mesa.)

A scalable record and its underlying unscalable record are two different types. Coercing the former to the latter (going into an implementor) requires that the sealed flag be set. Coercing the latter to the former is always possible -- the result is always sealed. The implementor can specify an arbitrary error check to be applied to a scalable record in order for the seal to be set: the check and the sealing take place atomically. This can be accomplished by copying the record, doing the check, and then atomically copying the checked result back. Alternatively, the record can be accessed through a level of indirection, and the indirect pointer can be disabled at the start of the check.

Miscellaneous

Filing

XLisp programs contain inter-program references (to dictionaries, in particular) that must be representable on external files, so that a program knows it will have the same environment when run subsequently, rather than some environment made up of objects that happen to have the same strings as their names.

We define a new object called a permanent name (PN) which consists of two parts: a file name (a string), which is to be considered only a hint, and a permanent id (PID, an integer of roughly 64 bits), which is the actual name of the object, and which is guaranteed different for every permanent object. Potentially, many different kinds of objects have permanent names -- function definitions, dictionaries, maybe any object whatever. To find the object addressed by a permanent name, it is necessary to hunt around in the file system, presumably starting at the file named in the PN, until an object with the right PID is found. Each file contains a directory that maps PIDs to text representations of objects in the file.

A PN is only needed for an object if a cross-file reference (or multiple reference within a single file) is made to it. PNs and PN directories are large, so we think it would be satisfactory to assign a PN only when (1) an object was written on a file, and "potentially referenced via a PN" was a property of its type; (2) a program asked for a PN for an object.

PNs are a basic mechanism for constructing permanent pointers. To make them useful, one needs such things as the notion of versions of an object, directories mapping PNs of original versions to PNs of current versions, etc. The crucial question is one of overwriting: how can one change an object without changing its PN? Fortunately most (all?) file storage systems provide some kind of atomic operation which is guaranteed to execute either completely or not at all, and an atomic overwriting operation can be constructed using this.

Lisp facilities needed in Mesa

We distinguish two kinds of facilities presently available in Lisp and absent from Mesa which are needed in an EPE Mesa:

> *Foundations*: basic, low-level facilities needed to match the functionality of Lisp: atoms, variable syntax analysis, dynamic variable binding, a safe language subset, and automatic storage deallocation.
>
> *Features*: equivalents for the Lisp Masterscope, file package, programmer's assistant, Helpsys and similar features, in a more general setting, together with improvements to the debugger and editor, facilities for specifying the construction of complex systems, and other "environmental" capabilities.

We propose to provide a few of these facilities, which we lump under *Preliminaries*, by initially constructing an interim integrated Mesa environment (IME) which provides quick edit-compile-run turnaround for the existing Mesa language and system, as well as rudimentary access to the Dorado's large virtual memory. This system serves as a base for later developments.

The following facilities from the ABC list are present in Lisp but not currently available in Mesa:

> (A1) Automatic storage deallocation
> (A5) Fast turnaround for minor program changes
> (B3) Run-time type system, self-typing data
> (B10) Run-time availability of all source program information
> (C3) Compiler/interpreter available with low overhead at run time
> (C6) Program-manipulable representation of programs

Two other Foundation facilities emerged from our discussions as being extremely important to the Lisp way of doing business:

> Dynamic variable binding
> Programmable syntax analysis (for both programs and data)

Of all these facilities, we will deal with (A5) under Preliminaries, and we think (C3) is covered by our proposals for (A5) and (B10); the rest are Foundations. We will deal with Features separately.

Preliminaries

This section describes the steps required to obtain an interim integrated Mesa programming environment (IME). The intended user community for this environment is a small corps of experts who will use it to construct a more lasting integrated Mesa world.

Phase I

Phase I represents the bulk of the effort. When completed, it will have produced an environment in which a text editor, the compiler, the binder, the debugger, and the user's program(s) are all simultaneously available. The facilities provided in Phase I are:

> Microcode and system extensions to support a 24-bit address space and virtual memory, with some crude protection between major entities (compiler, debugger, editor, user programs). Some of this work has already been done for Extended Memory Alto/Mesa.
>
> A window-based display interface for interaction with the debugger, editor, and other tools. Such an interface already exists in the Mesa debugger and in the Mesa Tools Environment, and one is planned for Chameleon, a package for market probes being developed by AOS.

A packaged text editor of about the functionality of the Laurel editor. The Tools group already have such an editor. Chameleon will have a similar editor. Alternatively, it wouldn't be much work to write our own.

Changes to the compiler and debugger to allow the symbol tables and program being debugged to reside in the address space for fast access.

Additional commands in the debugger to allow it to assume the role of the Alto Executive.

Phase II

Phase II will provide two extensions to the Phase I system:

Replacing an existing module by a new version without losing execution state. This involves changing the compiler to remember the decisions it made the last time it compiled the module, and is only possible if the structure of the module (global variables and entry procedures) has not changed. The loader must detect and invalidate extant frames that point into the global frame being replaced.

A dependency analyzer that determines what modules might (logically) require recompilation as a result of a change. SDD already has a version of this program called the Include checker.

Foundations

1) Programs as data

The ability to treat programs as data requires three major facilities within the Mesa environment: (a) an S-expression-like representation of source programs, (b) a means of using atoms (in the LISP sense) uniformly throughout the environment, and (c) a universal pointer mechanism.

S-expression representation. We propose to define a representation other than parse trees, so that the representation of parse trees can change as the compiler/language evolve.

Atoms. The Mesa compiler currently builds symbol tables that include a mapping between source identifiers and internal unique IDs (atoms) for an individual module. The Lisp atom capability requires a global map, into which the individual map would be merged when a compilation unit "enters the environment". This is very similar to the merge now done by the compiler for included modules.

Universal Pointers. We will need pointers that carry the type of their referent with them. This in turn requires a scheme for universal type IDs, which do not currently exist in Mesa. We also intend to add a facility for coercing a universal pointer to a pointer of more specific type; the coercion must be accompanied by a run-time check.

2) Programmable syntax analysis

LISP provides three opportunities for the programmer to affect the parsing/interpretation of input text: (a) at scanning time, (b) at compile time, and (c) at function call time. (a) may be accomplished by providing scanner escapes to user-written code, which will replace some appropriate substring of the current parsing context (i.e., the as-yet unclosed lists) with an S-expression. It appears straightforward to provide a facility comparable in power to LISP's readtables -- a more elegant mechanism might be desirable. (b) may be accomplished by more-or-less conventional macro expansion along the lines of the Mesa mechanisms for in-line procedures.

(c) is deemed undesirable and will not be provided.

Lisp allows free mixing of code and (structured) data, and parses both in essentially the same way; programs frequently read in both at run time. Consequently, the scanner and parser must be packaged for run time use.

3) Dynamic variable binding

Mesa, at present, only supports static binding of identifiers. To be able to bind a name to a variable in an enclosing *dynamic* context, we need to know what variables are to be found in each frame. Instead of the LISP default, we propose that variables to which dynamic binding may occur be declared as such at compile time. The compiler can then produce a table of entries of the form

atom -> <frame,type>

for each compilation unit. Dynamic binding would then occur much in the same way the signal-to-catch phrase binding occurs (but more efficiently).

4) Safe language

By a safe language we mean here one in which a pointer value of a given type POINTER TO T is guaranteed to point to a region of storage containing a value of type T.

There are three ways in which a pointer variable may acquire an improper value: lack of initialization, incorrect computation (e.g. by pointer arithmetic), or premature deallocation of the object pointed to. Array and string descriptors require the same treatment as pointers. In addition, it is necessary to do bounds checking on array references, since otherwise every array index is as dangerous as a pointer.

Preventing uninitialized pointers is straightforward, by generating code to initialize all pointers in procedure frames when they are allocated, and in dynamic variables of record and array types when they are allocated. Mesa 5 will have language constructs for allocating dynamic variables, so it is known where to do the initialization.

Preventing incorrect computation requires excluding all the dangerous constructs from the language

a) Loopholes into a pointer type or a pointer-containing type (PC for short)

b) Computed or overlaid variants in which the variant part is PC

c) Pointer arithmetic

d) Use of UNSPECIFIED (this restriction is probably stronger than necessary)

e) What is missing?

Preventing premature deallocation has two aspects: dynamic (explicitly allocated) variables and stack variables. For the latter, we propose to eliminate the ability to obtain an explicit pointer to a local variable, and as partial compensation introduce **var** parameters (call by reference). For dynamic variables some support from an automatic deallocation mechanism seems essential. We see three possibilities, discussed in the appendix: the one that comes closest to Lisp is fully automatic deallocation based on an incremental garbage collector.

Mesa currently has primitive and unsafe facilities for relative pointers, which there is a design to strengthen and improve. Misuse of a relative pointer can only damage data in the region it refers to; if none of the storage in this region contains any external pointers, then no global damage can occur. Embedded external pointers are still possible if they are treated as array indices, i.e. go through fingers. The advantage is that the programmer can forgo the safety precautions for such relative pointers without risking the integrity of the entire system. Of course a bounds check is still required (analogous to an array bounds check), but in many cases of interest it can be provided by storing the pointers in n-bit fields and making the size of the region 2^n.

5) Automatic deallocation

Our proposal is to have automatically deallocated (AD) types, and to use the Deutsch-Bobrow scheme for incremental garbage collection. This requires knowledge of the location and type of every pointer to an AD type, but the techniques are well-known and straightforward.

Features

This section attempts to characterize the "programmer's assistant" facilities of a Mesa-based EPE, in terms of a more or less direct replacement for existing Interlisp functions. However, we must bear in mind that we have the opportunity to redesign the user's interface to the system, taking the high bandwidth display into account. The goal will be to provide a consistent, integrated default interface for program development and for control of user applications that do not have exotic interactive requirements. The right facilities for such an environment will look substantially different from previous ones, as environments like the Smalltalk browser, DLisp, and the Mesa Tools are beginning to show us. These issues deserve a great deal of additional design work once the project is underway.

Interpreter

We advocate extending the compiler to provide the equivalent of incremental compilation for single statements in specified contexts. An evaluator within the debugger would generate the appropriate linkages to get the statements executed. Sufficient information would be saved to provide the History List features (below.) An appropriate default global context could be made available for the execution of statements corresponding to the Lisp "top level".

Break and Advise

Only modest extensions to the current conditional breakpoint code, using the incremental compiler invented just above, are necessary to obtain a break/advise facility the equal of Interlisp's. The standard breakpoint handler could take care of providing the linkage to the system-produced procedures containing the advice.

History Lists and their Use

A history list that records top-level user actions in external (text) form, along with an indication of the contexts in which they were executed, are sufficient to implement the *Redo* command and the function-composition capabilities that it enables. To provide *Undo*, however, requires internal information as well. Once automatically allocated lists, programs as S-expressions, atoms and data with attached types, and the other *Fundamentals* are available, implementation of both commands should be straightforward. Probably both would be done, as in Lisp, by retaining S-expression representations of the original statements and their reversing functions.

In Mesa even simple programs consist of several independent global contexts. Virtually every user action will have to be interpreted with respect to a specific environment. History list entries will have to include the corresponding environmental information, and the commands will have to know when the corresponding environments no longer exist, or will have to arrange, as Interlisp can, to retain the context frames as long as they are referenced (requires garbage collector to operate on frames.) Contexts running as concurrent processes could also cause trouble if more than one of the processes becomes involved in user history activity. The history provisions would want to be more fully integrated with the complex context structures than are the Interlisp models.

Implementating the equivalent of the TESTMODE(T) facility in Interlisp -- saving the old contents for *every* store other than into local variables -- could cause both time and space problems, because Mesa is much more an assignment-based language than Lisp.

Given Mesa's complex syntax, it is not clear how best to provide undoable versions of each language

primitive that can produce a storage modification (e.g., a /← b ?). Languages like Alphard provide a model for how this might be done, by regarding infix operators as abbreviations for function application, but this also requires eliminating the special role of primitive generic functions in Mesa. More generally, the maintenance of a syntax that is acceptable to both people and LALR machines will present problems throughout the system that are not encountered in the pure Lisp implementation.

Editor

If the editor chosen for the *Preliminaries* implements an undo/redo capability, integration with the more general version should be straightforward. Otherwise, these commands would have to be built from scratch, possibly at considerable cost (depending on the editor design.)

In a display-based editor, structure-oriented operations can be largely implemented as part of the selection mechanism. To the extent that the text files are in communion with the underlying S-expression representation, it will be easy to produce the needed hierarchical selections.

Dwim

Some spelling correction and the repair of minor typographically induced syntax errors could be incorporated into the compiler. The user interface to these tools are likely to be quite different in a system based on compilation and display editing.

MasterScope

The structures needed for the *Fundamentals* implementations are adequate for the development of a Masterscope-style database. An extension of the interface used for the other tools would provide user access. The compiler, binder, or an intermediate agent, would notice change and update the data base.

Character of result

We have agreed that very similar results could be obtained starting from either Lisp or Mesa. We also agree that the EPE must accommodate a range of styles including the present Lisp, Mesa, and Smalltalk styles, and our goal is to support these styles without prejudice. However, we expect that a variety of influences will cause both the intermediate and final results to differ depending on the starting point, primarily in terms of the efficiency with which they support early versus late binding. The following two sections set forth what we think the "flavor" of a Lisp- or a Mesa-based EPE would be -- the strengths (and weaknesses) that would result from starting with each of the two possible base languages.

Lisp-based

Aside from the AI effort, a number of other projects in the lab (such as the original wirelister, the Dorado timing analyzer, the PARCHIVE program) have been implemented in Lisp. Perhaps the single most important observation about the use of Lisp is that as users become more experienced, they start building tools within the system to help them. These tools may span many projects, or they may be very specialized; they range from complete languages (KRL) to minor extensions (automatic checking for undefined functions when one leaves the editor). The AI group in particular has made heavy use of this ability to extend the Lisp system: Lisp has been uniquely suited to their approach, which is to solve problems by first building a new language in which to express significant parts of the solution, and then building the solution (simultaneously extending the language). The Humus effort within Mesa offers a contrasting example of an attempt to build an extension on a system that did not support it well: Juniper, for example, probably could have benefited substantially from some extensions of this kind.

It is important that an EPE support the working style which builds tools and even languages as part of the problem-solving effort. Lisp supports this style well in part because it is:

Incremental: Small changes require only a small amount of work (both mental effort and real time). This is true for both ordinary Lisp programs and programs written in embedded languages. (Current implementations, unfortunately, have relatively weak tools for discovering whether one's changes are consistent.)

Open: The basic facilities of Lisp are open to change at a very low level: one can modify the operation of the entire system from a user program. There is no distinction between system and user code, variables, name spaces, etc. (This is also a weakness, in that it is not uncommon to find that parts of the system make assumptions about each other that casual modifications violate.)

Integrated: The user can slip relatively gracefully from procedures written in an embedded language back to procedures written in Lisp itself, and vice versa. Since the parser and interpreter are packages, code (in a variety of languages) can be stored in data structures and executed when retrieved.

Aware of user activities: Standard packages in Lisp keep track of new objects added to the system by the user on-line, and changed objects, and help the user keep track of his complex environment. Because all transactions with system objects (such as editing, creating new variables, etc.) are handled through an active intermediary and a set of functional interfaces, it is easy to provide all the "hooks" for complex assistants like Masterscope. .

Abstraction-based: The user is completely freed from concern with basic questions like the representation of integers and symbols, and management of storage. (Most Lisp implementations, including the present Interlisp, have a related weakness, in that they provide few mechanisms for attaining better efficiency even when the user knows full generality is not needed.)

Mesa-based

The "Mesa style" tends to place greater emphasis on structure than on unconstrained flexibility. Probably its two most important aspects for an EPE are its emphasis on static checkability and its provision for explicit interfaces. Both are important in speeding up the programming process and in improving the quality of the result; they become even more so if the units that programmers manipulate are large (packages or subsystems), rather than small (statements or functions); the advantages will be small for programs whose "characteristic times" (design, programming, checkout, existence, total execution) are all measured in minutes, large if they are measured in weeks or months. In an environment where programs are undergoing rapid change, however, mandatory checking mechanisms tend to introduce unnecessary overhead by requiring complete internal consistency at every step of the development process.

Static checking: The present Mesa style encourages (and often requires) relatively early binding of many aspects of programs that in Lisp are typically bound during execution. Mesa also incorporates redundant information (e.g., declarations) that can be checked for consistency. These properties make it possible to routinely check statically for program faults that in Lisp would normally only be detected at run time. (A good example is Mesa's type-checking.) It is generally better to locate faults by static checking, because

> Many faults can be identified in a single run of the checker, rather than surfacing one at a time in debugging runs.
>
> There is experimental -- as well as anecdotal -- evidence for the proposition that program faults located statically are diagnosed and removed more quickly than those located dynamically.
>
> Passing the static test ensures that *all* faults in a given class have been removed; in general, no finite set of test cases gives such assurance with dynamic checking.
>
> "Correctness" is a static property of the program text; it is hard to ensure that a program that relies heavily on dynamic properties actually does what is intended.

A Mesa-based EPE would make provision for binding at a variety of times, but delayed or dynamic bindings would occur at the programmer's request, rather than by default. Tools would also be available to exploit program redundancy to *infer* suitable declarations for programs written without them, and otherwise make it easy to convert programs originally written in a delayed-binding style to earlier bindings. (It should be noted that such tools have already been developed, experimentally, in a Lisp-like environment.)

The belief that this style actually speeds programming (measured as problem solutions per unit time) is intimately tied to the observation that most programmers spend more time worrying about the possibility of programming errors -- and coping with their consequences -- than they actually spend writing new code. This style does require more planning before a running program is created -- some would consider this a disadvantage. A definite weakness of this approach is that it will be relatively difficult to add flexibility that was not anticipated in the program design, thus restricting the range of experiments that can be performed easily.

Explicit interfaces: Much of the effort in the design of Mesa went into the development of facilities for a modular style of programming. Definition modules are program components used to specify interfaces between clients and implementors of abstractions -- permitting them to work independently, and make changes independently, as long as they respect the interface. (Ideally, they would completely liberate each side from any knowledge of the internals of the other, but Mesa has not quite attained this ideal.)

It seems quite clear that one of the major thrusts of our future research must be building systems out of standard components, and decoupling the implementors and users of such components; Mesa provides our best effort so far in this abstraction-based direction. The need to specify interfaces in advance may be cited as either a strength or weakness of the Mesa approach, depending on circumstances and prejudices. The present need for vast recompilations whenever a fundamental

interface is changed is a weakness that can certainly be reduced, and probably eliminated by incremental recompilation on demand.

Other notes:

A Mesa EPE would allow programming to be, in a very real sense, incremental, because much more of the burden of propagating the consequences of any change to a program could be assumed by the compiler and other EPE tools; conceptually small changes would not generally require large amounts of reprogramming.

One happy consequence of relatively early binding is that decisions made statically are generally much less expensive than those made (repeatedly) dynamically. *This consequence should not be interpreted as the motivation for encouraging early binding in the EPE.* Some of the present efficiency of Mesa would definitely be sacrificed in those situations where greater flexibility and later bindings are introduced. However, it should not be hard to retain an "efficient subset" (roughly SDD Mesa).

Effort and schedule -- Lisp base

A Lisp-based EPE (here called XLisp) can evolve from our current systems in a series of steps, with a working (although not stable) system after each step. The steps listed below need not be done in exactly the order given, although we believe this order leads to the least total work. We also believe little parallelism is possible *between* any two of the steps, although, as the assignments of people indicate, a lot of parallelism occurs *within* a single step.

The names attached to steps other than the first should not be taken too literally -- the named people have not been consulted. We have assumed the worst, namely, that almost no volunteers from outside the current Lisp community will participate in this work.

The total of the figures below is: 87 work months, 27 1/2 elapsed months. This total does not include work on Masterscope, but we assume that this work will be done eventually by Larry Masinter, who is not currently available. At the end of this period, we have a system which includes all the present Lisp facilities plus the critical Mesa-like facilities discussed earlier in the Facilities section.

The memo on which this section is based, which contains a fuller discussion of the sequencing and rationale for the various steps, appears in the appendix to this report.

Preliminaries

Dorado Interlisp

The first step is to get Alto Lisp running on the Dorado. This work began in early November and is well underway. For more details see a memo by Bobrow, [Ivy]<Bobrow>dlisp-status.memo.

After this step we have a usable Interlisp system running on the Dorado.

Work months: 18
Elapsed months: 6?

People: Bobrow, Deutsch, Haugeland, Teitelman; Fikes, Goldstein, Kaplan, M. Kay; Laaser, Masinter, Thompson

Main effort

Shallow binding

Change the system to use shallow binding and have invisible indirection for variables.

This step produces no externally visible changes in the system, except that GLOBALVAR declarations are no longer necessary.

Work months: 3
Elapsed months: 1

People: Bobrow (stack fns & interpreter), Deutsch (compiler & loader), Haugeland (microcode)

Clean up macros

Change the way macros and special syntactic forms are handled, to eliminate "error-driven interpretation". In particular, eliminate DWIM corrections that require altering or examining the state of the interpreter.

We can arrange things so that after this step the system will still run existing Interlisp programs, but MACRO properties and NLAMBDAs will produce warning messages. On the other hand, packages like the iteration and pattern match statements will fit into the system much more cleanly.

Work months: 6
Elapsed months: 1

People: Bobrow (interpreter), Deutsch (compiler), Teitelman (DWIM/CLisp), any 3 Lisp programmers (scan the system); defer work on Masterscope

Dictionaries

Add dictionaries to the system, but not lexically-scoped closures. The hardest part of this step is probably designing the proper interfaces to dictionaries-as-virtual-objects.

After this step we have a system which supports localization of names. Individual packages can be rewritten to take advantage of this at any time. Some lame-duck code must remain in the file package to be able to dump and load individual definitions in a global dictionary so that existing programs will run.

Work months: 15
Elapsed months: 2 (design), 3 (implementation)

People (design): Bobrow, Deutsch, Masinter, Horning/Lampson

People (implementation): Bobrow (stack & interpreter), Deutsch (compiler), Fikes (record package), Kaplan (dictionary objects), Teitelman (file package & miscellaneous); defer work on Masterscope

Closures

Implement closures. This includes lexically scoped FUNARGs, and arbitrary closures like Mesa module instances. Careful design is needed here to produce reasonably efficient results.

After this step the system will support instantiable packages and Smalltalk-style object-oriented programming. Old programs are not affected; however, some existing packages should be recast in instantiable form, and there will be a period of confusion and trauma while this happens.

Work months: 9
Elapsed months: 1 (design), 2 (implementation)

People (design): Bobrow, Deutsch, Levin

People (implementation): Bobrow (stack functions), Deutsch (compiler), Haugeland (microcode), Teitelman (BREAK et al)

Type calculus

Implement the type calculus (types as objects, type checking), but entirely on a dynamic basis -- no variant representations, and no compile-time checking.

The result of this step is to make the type system a first-class citizen, rather than an intermittent visitor as in the Decl package. Programmers can start writing type declarations in the knowledge that they will reap the benefits after the next step.

Work months: 9
Elapsed months: 1 1/2 (design), 2 (implementation)

People (design): Bobrow, Deutsch, Horning/Lampson, Morris, Scott?

People (implementation): Bobrow (interpreter), Deutsch (compiler), Kaplan (type objects)

Compile-time type checking

Implement compile-time type checking.

This step gives us a system that has nearly all the functional capability of a "safe" Mesa, although it is considerably less efficient in its representation of some kinds of objects, e.g. bounded integers and set elements. It is not clear what to do about the Binder.

Work months: 12
Elapsed months: 3

People: Deutsch, Levin, Satterthwaite (compiler), Kaplan (PUTD, APPLY, etc.)*

Variant representations

Implement variant representations, such as unboxed numbers.

This system should be competitive with Mesa, aside from the questions of LOOPHOLEs, pointer arithmetic, etc., and of course it has a lot of capabilities not in current Mesa.

Work months: 6
Elapsed months: 1 (design), 1 (implementation)

People (design): Deutsch, Satterthwaite

People (implementation): Bobrow (stack functions), Deutsch (compiler), Fikes (record package), Haugeland (microcode)

Further work

Infix front end

Implement an infix front end, i.e. an infix parser and prettyprinter.

Work months: 3
Elapsed months: 1

People: Goldstein (prettyprinter), Swinehart (editor), Teitelman (integration)

Permanent pointers

Implement permanent pointers. Permanent pointers, as part of a programming system, require some research. Integrating them into the system will be an ongoing project.

Work months: 6?
Elapsed months: 2?

People (design): Deutsch, Horning, Sturgis

People (implementation): Sturgis (primitives), Teitelman (file package)

Effort and schedule -- Mesa base

As discussed under Facilities above, the plan for a Mesa-based EPE has three major stages: Preliminaries, Foundations, and Features. The stages must be done in approximately the indicated order, although some overlap is possible. An important property of the plan is that growth of the system is highly incremental: even within a stage, there are many points at which a stable system with improved function is available.

This section of the report is a condensation of a memo submitted to the working group. For more detail, see the full memo in the appendix to this report, and the earlier discussion under Facilities.

Preliminaries (IME)

Phase I

Phase I represents the bulk of the IME effort. When completed, it will have produced an environment in which a text editor, the compiler, the binder, the debugger, and the user's program(s) are all simultaneously available. The components of the Phase I system are listed below with estimates of the effort required to produce them.

Approximate Effort to Construct Phase I System: 9.5 work-months

Microcode

Extend the microcode to handle long pointers and page faulting. *Approximate effort*: 1 work-month. (Note that this work will be done regardless of IME plans.)

System

Change the Alto/Mesa runtime environment to handle page faults and support (code) segment allocation in the 24-bit address space. Nearly all of this work has already been done for the Extended Memory Alto version of Alto/Mesa. *Approximate effort*: 1 work-months.

Provide rudimentary protection through separate MDSs for major entities. *Approximate effort*: 2 work-months.

Interactive Base

Acquire a window-oriented display manipulation package, from either the Mesa debugger or the Tools Environment. *Approximate effort*: 0 work-months (That is, we'll take what they give us.)

Alternatively, acquire the Chameleon package from AOS; this would supply a data base package and a text editor as well (see below). *Approximate effort*: 0.5 work-months. (That is, we won't adopt this package unless it appears stable and we can use it with only minor tweaks.)

Text Editor

One possible source for an adequate text editor (i.e. about as good as the Laurel editor) is an existing editor that runs in the Tools environment. *Approximate effort*: 1 work-month.

Chameleon is an alternative source for a comparable editor. *Approximate effort*: 1 work-month.

(The effort estimates above are intended to be upper bounds. Butler believes we can "hack together an editor in 2 work-months", so we probably shouldn't invest more than 1 work-month in

some other package.)

Compiler

Maintain a cache of symbol tables in the virtual memory space. *Approximate effort*: 2 work-months.

Debugger

Extend the debugger command language to include editing, compilation, binding, etc. Also change the debugger to use the common symbol table cache. *Approximate effort*: 1.5 work-months.

Binder

We intend to make no functional changes to the binder. *Approximate effort*: 0.5 work-months.

Phase II

Phase II will provide two extensions to the Phase I system. These extensions are independent and could be performed in either order.

Approximate Effort to Construct Phase II System: 4 work-months

Replacing an Existing Module

Make the compiler retain more information about the structure of a compiled module, so that a slightly changed module can often (almost) be slipped into the system as a replacement for the old version without altering the current state. *Approximate effort*: 2 work-months.

Change the loader to find and alter extant frames affected by loading a new version of a module. *Approximate effort*: 1 work-month.

Understanding Module Dependencies

Implement the DeSoto dependency analyzer, which ensures that all modules that must be recompiled will actually be recompiled. *Approximate effort*: 1 work-month.

Foundations

As discussed under Facilities above, there are five major basic language/system facilities which need to be added to existing Mesa. These facilities comprise the Foundations stage of the Mesa-based EPE work.

Approximate total effort for Foundations stage: 20 work-months, including 4 work-months for cleanup of the current Mesa compiler. These estimates result from consultation with Ed Satterthwaite for compiler-related things, and with Peter Deutsch for things copied from Alto/Dorado Lisp.

1) Programs as data

S-expression representation. Approximate effort: 1 work-month.

Atoms. Approximate effort: 1 work-month.

Universal Pointers. Approximate effort: 4 work-months.

2) Variable syntax analysis

Implement programmer-controlled parsing at input time, using the existing Lisp read-table design. *Approximate effort*: 3 work-months.

Implement programmer-controlled extension at compile time, using macro expansion, based on Mesa in-line procedures. *Approximate effort*: 1 work-month.

3) Dynamic variable binding

Make the compiler produce a table of entries of the form

atom -> <frame,type>

for each compilation unit, and use this to implement Lisp-style dynamic variable binding. *Approximate effort*: 1 work-month.

4) Safe language

Prevent uninitialized pointers and array descriptors. Do bounds checking on arrays. Exclude dangerous constructs (e.g. LOOPHOLE) from the language. Replace @ on local variables by var parameters. Replace explicit freeing by deferred freeing or automatic deallocation. Make relative pointers safe. *Approximate effort*: 2 work-months.

5) Automatic deallocation

Implement the Deutsch-Bobrow scheme for incremental garbage collection. This requires knowledge of the location and type of every pointer to an automatically deallocated type. *Approximate effort*: 3 work-months, including necessary compiler changes.

Features

This section attempts to estimate the cost of the "programmer's assistant" facilities of a Mesa-based EPE. Here we consider the activities required to provide a more or less direct replacement for existing Interlisp functions. As discussed in the Facilities section above, we do not believe that this is the best approach to providing these facilities in the EPE.

We assume that the work described under *Preliminaries* and *Fundamentals* has been completed. The text editor and debugger described in those sections, perhaps augmented by Mesa Tools, would certainly provide an adequate interface. The time estimates are presented in the same spirit of optimism that pervades the rest of this document.

Interpreter

Extend the compiler to handle single non-declarative statements in specified contexts. An evaluator within the debugger would generate the appropriate linkages to get the statements executed. Save sufficient information to provide the History List features (below). *Approximate effort:* compiler extension: 2-4 work months; evaluator, top-level environment: .5-1.0 work months. If all or part of this work is implicit in providing the "programs as S-expressions" fundamental capability, the estimates can be reduced accordingly.

Break and Advise

Extend the Mesa debugger's current conditional breakpoint code to obtain a break/advise facility at least as good as Interlisp's. *Approximate effort:* 1-3 work months.

History Lists

Implement Undoing, as in Lisp, by retaining S-expression representations of the original statements and their reversing functions. Integrate history facilities with Mesa's more complex environment structure. Add language syntax for undoable operations. *Approximate effort:* 2-4 work months to implement the mechanism; development of primitives and system procedures with undoable versions as time goes on.

Editor

If the editor chosen for the *Preliminaries* implements an undo/redo capability, integration with the more general version should be straightforward. Otherwise, these commands would have to be built from scratch, at considerable cost (depending on the editor design). *Approximate effort:* 1 work month or ? work months.

Implement structure-oriented operations in the editor. *Approximate effort:* 1 work month.

Dwim

Incorporate some correction of minor spelling and typographically induced syntax errors into the compiler. *No estimate.*

Masterscope

Extend the interface used for the other tools to provide user access to a Masterscope-like data base. Put code in the compiler, binder, or an intermediate agent to notice changes and update the data base. *Approximate effort:* the data base facilities, no estimate; the interface, 2 work months.

Participating people

All the people listed at the beginning of the report are candidates for participation in the actual EPE effort, and most of them have indicated a willingness to contribute in some way regardless of whether a Lisp or Mesa base is chosen. However, nearly all of these people have substantially more experience in one of the two languages than in the other. To achieve our goals, it is essential that people with both kinds of experience participate in the project in a major way: we are very concerned that if a Mesa base is chosen, people with Lisp backgrounds might not feel highly motivated to contribute their unique experiences and insights to the project, and conversely if Lisp is chosen.

The following are individual responses to a message we sent out to everyone who had indicated an interest in the programming environment area by putting themselves on an on-line distribution list. The absence of someone from the following list does not mean they are not interested in working on EPE.

Birrell

I would probably be interested in low-level work in EPE (i.e. the parts which might be considered as an operating system), and in at least keeping track of language design.

Bobrow

If it is a Mesa based EPE, I expect to put about a third of my time in on it. If it is a Lisp based EPE I expect to put in about 2/3.

DaveSmith

I cannot participate in actual implementation design or coding. I guess that puts me in the class of kibitzers. However I have several ideas on programming languages and programming environments that I would like to share. (Most of these ideas came out of the LISP70 project that Enea, Tesler and I did at Stanford.) I am anxious for EPE to succeed because PARC is not currently at the state-of-the-art in programming environments. In the short term, some simple extensions would benefit us all (such as Lampson's and Levin's intermediate Mesa PE ideas). But in the longer term, I hope EPE turns out to be a laboratory for basic research into programming languages and techniques. I think there's at least a 5-10 year project waiting there.

Deutsch

I expect to spend about half my time on EPE through mid-1979. What happens after that depends a lot on how well we are doing at genuinely incorporating the strengths of all three languages into a single system. If things are going well, I expect EPE to continue as my largest single interest for at least another year after that.

Gifford

I am interested in the tradeoffs between protection mechanisms and type systems for building layered systems; I am especially interested in the design of a flexible protection scheme for the D* machines that would enable the construction and debugging of large systems. This interest also encompasses hardware/microcode support for a run time type system. I strongly favor Mesa over Lisp as the EPE base.

I plan to spend the first part of January reviewing the EPE material in detail. It may be possbile to view part of my work in the comming year as EPE related, or my work as part of the EPE project, or both; it depends on the intersection of my thesis interests with EPE needs.

Guibas

Here is a very selfish reply. At this late hour it is perhaps best to paint matters as black or white, rather than gray.

An (incomplete) list of epe areas that interest me is:

arithmetic, text formatting, graphics, pattern matching, and searching.

My motivation is twofold: to have a friendly environment in which to do my own research, and to make sure certain fundamental facilities are provided at an acceptable level of quality.

My time commitment would depend heavily on my research interests and the congeniality of epe to my working style. For example, a very important attribute for me is conciseness of notation. Has it been a serious epe desideratum?

With regard to base language choice, my deep-down feeling is that a lisp based environment would be infinitely more appealing to me than a mesa based one.

Then again, don't underestimate the power of persuasion...

Horning

I consider myself committed to EPE as my major technical effort for the next couple of years, independent of starting point. My background implies that my contribution to early phases of the project will depend on the starting point: with Lisp, I would expect to contribute to requirements definition; with Mesa, I could get involved in the system programming.

Lampson

I expect the EPE to be my major research interest for the next 18 months. There is some chance that I will change my mind about this if Lisp is the base. I expect to be able to spend a lot of time on design, but am less optimistic about begin able to do any substantial programming jobs.

Levin

I consider that, if EPE is Mesa-based, I will consider it my second-priority project (after the distributed transport mechanism). That translates to about 30% of my time over the next year. After that, I can't say. I would expect my involvement with EPE to be greatest during the earlier phases, when fundamental structures are being built. I probably couldn't contribute much to a Lisp-based EPE.

Masinter

I plan (if I am hired by CSL) to devote full time to EPE starting July 1 working on those pieces of system which are prerequisite to building Masterscope-like facilities for Mesa. I imagine that actually getting to actual Masterscope facilities will be in the rather distant future: working on either an S-expression representation (easier to analyze), data base facilities (shared data base server), access to remote files, etc. will take first priority.

Morris

I might sign up to work on the design and implementation of a more powerful static checker for the language. I guess about half-time would be my level of commitment.

Satterthwaite

If Mesa is chosen as the base for CSL's EPE, the implications for SD will be carefully considered. We have no current position on the numerous policy questions that would be raised by direct participation, but contributions to subprojects of mutual interest seem likely to the extent that SD's goals and schedules permit.

My principal interests re the EPE issues are (a) designing a "strong" but much more flexible type system, (b) (for Mesa) designing a safe language subset and storage reclamation techniques, (c) implementation of (a) and (b).

Suzuki

Database:

There are two issues concerning database in EPE. One is the database for interaction, namely masterscope level thing. I would like my database system to be considered for the use in EPE system along ASD and masterscope. I will demonstrate the system tomorrow so many of you may have a chance to evaluate. The other is the access from the programmers. This may be some type extension and packages. If the previous line is chosen I will spend quite a few time of mine devoting bringing up the user interface. If not I will be interested in the latter subject which I would be interested in spending some of my time.

Basic microcode:

I am willing to support Dorado Mesa and its extension. Birrel may want to do this in connection with Pilot. In that case I am willing to help him.

Subrange checking, verification, etc.

Probably not much to do for a while.

Compiler, etc.:

I have quite a few experience in writing compilers. But there are so many people who can do it, so unless there is some urgent need I may not particularly want to do it.

Swinehart

I'm willing to play a substantial role in what you've called the Tools segment of the EPE project. I cannot in all honesty call it my primary activity, however, since it seems extremely important to further the voice communications work. Percentages are probably misleading, but I'd expect to devote 30-50% of my time to it. Goals: unified (default) approach to user interface for program development, control of applications.

Taft

I expect to spend a majority of my time on the Laurel transport project during the next 6 to 9 months, so I can commit only a fraction (say, 25%) until that is done, but perhaps more later (50 to 75%).

As for what I might do: if the EPE is Lisp-based, one obvious task would be to implement a Pup package. If it is Mesa, I am less certain what to do, though one possibility is to extend the file system to access IFS or Juniper remotely. I am also prepared to contribute effort in whatever other ways seem appropriate, e.g., writing Dorado microcode. Since I am not familiar with the insides of either Lisp or Mesa, and I am only superficially familiar with Lisp as a whole, it is probably not appropriate for me to work on the basic kernel of EPE.

Teitelman

I expect significant portions of my time will be involved in epe, regardless of which base they start from. If mesa, then would be in implementing various user level packages not in mesa, such as programmer assistant, file package, dwim type stuff, etc. If starting from lisp, I have a less clear

picture of what I would be doing, at least initially, since a lot of things have to be changed down in the guts of lisp, but I would think I would be fairly involved simply because I am currently custodian of so much of the system, and can also make inputs about how various changes will affect the style of lisp programming.

Project organization

The working group did not spend any time on organizational questions. However, we identified two areas in which people would be needed almost immediately, and noted the possible relevance of the current work on Dorado Lisp. In addition, a 3-person coordinating committee for the EPE project has been appointed by CSL management (Bob Taylor). This committee (Deutsch, Horning, Lampson) has met briefly and produced some organizational ideas described below.

Coordination with SDD

If a Mesa base is chosen, we will need someone to function as a continuing liaison with SDD. This person will need to be familiar with the language and system work being done in both places, to bring both potential divergences and opportunities for sharing code to the attention of the appropriate people doing the work. We believe that our goals and SDD's are similar enough, at least at the language level, that such a liaison could be very helpful.

IME (Integrated Mesa Environment)

As mentioned in the earlier section on Effort, we believe it is worthwhile to make a short-term investment in improving certain gross aspects of the current Mesa environment, if a Mesa base is chosen. We view this as an initial phase of the EPE (even though much of the code is likely to be superseded later) and hence it should be coordinated with the rest of the project.

Dorado Lisp

The current project to produce an Interlisp system for the Dorado is independent of EPE. However, if a Lisp base is chosen, the starting system will be Dorado Interlisp. In this case it will be to our advantage for people with EPE interests to become involved in the Dorado Lisp project sooner rather than later. We also expect that in this case the current CSL Lisp community will be the initial core of the EPE design and implementation effort.

Proposal for EPE project organization

We (the coordinating committee) believe the project should be organized into three areas: language/system, tools, and packages. Naturally these areas overlap; one of our functions is to facilitate communication among them.

In the language/system area, we believe the work will demand a fairly small, closely knit group of 3-4 people to work on the compiler and run-time system. At least one person from the coordinating committee should be a member of this group.

In the tools area, we believe that more autonomy of individual projects is possible, although certain decisions with system-wide consequences (such as display management, low-level hooks for programmer assistance, and data base interfaces) must be made at a global level.

In the packages area, we would like benevolent anarchy to prevail, subject to some planning for things we know will be widely used throughout the system (e.g. network communications) or require special expertise (e.g. high-quality mathematical routines). Our idea is to set up a standards committee, which would develop and publish a set of criteria that would be applied to all packages submitted for inclusion in the official library. Packages would be read by the committee for agreement with these standards, and by one other person (not the implementor) for appropriateness of design and implementation. Since this procedure would introduce a certain amount of inertia into the system, we envision the existence of an unofficial library, with the understanding that a package is only allowed to exist in this library for a limited amount of time (say six months), and that anyone who uses the package during that time may have to rewrite some code if the package's interfaces change as a result of the approval process.

Foreign affairs

The choice of Lisp or Mesa as the starting point for an EPE project within CSL has implications for other projects within CSL, for other organizations, and for CSL's relations with those organizations. Although we are agreed that "foreign affairs" should be given less weight than CSL's own needs, they are a legitimate concern in making the decision, and may prove to be one of the balance tippers.

The following discussion enumerates many of the areas of concern and indicates the probable effect of either decision. To a high degree, the effects are "formally symmetric," i.e., for every advantage of one choice, a corresponding advantage of the other choice can be found. However, CSL may wish to give higher weight to relations with the rest of Xerox than to its relations with the world outside Xerox. For intra-company relations, somewhat more advantages follow from the choice of Mesa.

Separate sections discuss the effect of the decision on the importation and exportation of people, ideas, and code, and on CSL's "image."

Importation

Importation of ideas

From within CSL

Issue: There is a lot known about programming environments within the lab, and people who are not directly involved in the project are still potential sources of good ideas.

If Lisp is chosen: The people currently working in and on the Lisp environment will likely be within the EPE project. Some effort will be needed to ensure that Mesons concerned with interfaces, static checking, verification, and optimization provide enough input to ensure that they can profitably build on the EPE as it becomes available.

If Mesa is chosen: The Mesons mentioned above will likely be within the EPE project. Some effort will be needed to ensure that the Lispers concerned with programmer assistance, programs as data bases, and integrated sublanguages provide enough input to ensure that they can profitably build on the EPE as it becomes available.

From within Xerox

Issue: Other parts of Xerox, most notably SDD, are also concerned with developing programming environments (although with somewhat different characteristics than EPE requires). We should be prepared to shamelessly steal good ideas.

If Lisp is chosen: The relationship to SDD efforts will be somewhat more distant, the application of their ideas somewhat less obvious.

If Mesa is chosen: Much of what SDD does will be directly relevant; there will be continual pressures to remain compatible, and to import code, rather than just ideas.

From the world at large

Issue: We would like to track the state of the art in the world at large.

If Lisp is chosen: Much of the "automatic programming" and "programmer's assistant" type of work is done in various Lisp dialects, and should be easily applicable. This group of researchers is currently being encouraged to focus their attention on the development of a programming environment for ADA (formerly DoD/1); however, much of their research work is likely to

continue to be in Lisp.

If Mesa is chosen: Much of the specification and verification work is directed towards Pascal dialects, and should be easily applicable. ADA, like Mesa, is based on Pascal, and work on ADA environments should be highly relevant.

Importation of people

Issue: We want people to move into the project easily, and are even more concerned that it be easy for users to convert to the *use* of the EPE.

From within CSL

Lisp and Mesa have roughly comparable numbers of "hard core" users, and the issues of migration seem roughly symmetric.

From within Xerox

Mesa will be much more widely known and used within the corporation.

From the world at large

Generic Lisp is more widely known than Mesa; Generic Pascal is more widely known and used than Interlisp. If there is a systematic difference, it probably lies primarily in the communities from which we expect to recruit.

Importation of code

From within CSL

Issue: Existing service packages should be made compatible with, or absorbed into, EPE to the greatest extent feasible. We want to avoid fragmentation of both user and maintenance effort. (In addition, importing people implies importing their research systems.)

If Lisp is chosen: The Interlisp packages come over almost intact. Will Bravo and Laurel users accept them as substitutes, or will functional equivalents of those systems be needed?

If Mesa is chosen: Provision of functional equivalents of the Interlisp packages will be a major chore. Laurel comes over, but something would still need to be done about Bravo.

From within Xerox

Issue: We want to stand on the shoulders of SDD and ASD wherever possible, both in terms of service packages/systems, and program development aids. CSL will want to experiment with prototypes and systems built elsewhere, and would benefit from the ability to incorporate outside packages and subsystems into our experimental systems.

If Lisp is chosen: Nothing comes over.

If Mesa is chosen: The amount of actual code that we can import depends both on the degree to which we force our EPE Mesa to remain equivalent to SDD Mesa, and the extent to which the solutions they choose to provide match the problems we need solved. Transferability will probably require that EPE Mesa be a strict superset of SDD Mesa.

From the world at large

Issue: It would be nice not to have to replicate large system-building projects from elsewhere.

If Lisp is chosen: There is significant precedent for Interlisp system facilities being imported from elsewhere. The amount of actual code that we can import in the future depends both on the degree to which we force our EPE Lisp to remain equivalent to Interlisp, and the extent to which the systems built elsewhere solve the problems we have. There is apparently relatively little history of importing application-level code in Lisp.

If Mesa is chosen: No code comes in.

Exportation

Exportation of ideas

To the rest of CSL

Issue: We want to "talk their language," even before conversion to EPE is complete.

Again, the Lisp and Mesa situations seem quite symmetric; special efforts at communication will be needed in either case to avoid alienation.

To the rest of Xerox

Issue: We would like to be able to demonstrate feasibility of ideas, and let the development groups take them over and produce the production systems.

If Lisp is chosen: The credibility gap will be somewhat larger, particularly if adequate performance of a demonstration system depends on the full power of the Dorado.

If Mesa is chosen: We will be under continual pressure to export not just ideas, but code, and then to maintain that code for the whole Xerox world.

To the world at large

Issue: As part of the commerce of ideas, we must give as well as take. (See also the discussion under importation of ideas.)

If Lisp is chosen: We communicate easily with the AI and theorem-proving communities.

If Mesa is chosen: We communicate easily with the programming language, system implementation, and structured programming communities.

Exportation of people

To the rest of CSL

Issue: None of us sees EPE as his life work.

However, regardless of starting point, we expect EPE to be *the* major programming tool within CSL for some time to come; there should be no major problem moving from EPE to any other CSL programming project.

To the rest of Xerox

Issue: One of the best ways to transfer ideas is often to transfer the people who carry them. We would like to make (temporary) transfers easy and productive.

If Lisp is chosen: In addition to the problem of moving to a less powerful programming environment, there will be a definite (but, one hopes, temporary) linguistic gap of the same order of magnitude as the effort required within CSL to convert a Lisper to a Mesa-based EPE. This will significantly increase "swapping overhead."

If Mesa is chosen: The transfer may be almost too easy; we need to ensure that we get people back!

To the world at large

Issue: Do we really want to encourage this, except on a temporary basis?

As with importing people, this largely depends on the communities with which we want to interact most strongly.

Exportation of code

To the rest of Xerox

Issue: Some systems that we build primarily for our own use may prove to be so good that we want them used widely within the corporation (e.g., Bravo); in other cases we may need a large community of users to enable our own research (e.g., message transport system).

If Lisp is chosen: This won't happen, *unless* provision is made for "severable" systems (analogous to Mesa image files) and programs can be tuned to run with acceptable efficiency on D0's.

If Mesa is chosen: This depends on degree of source-language compatibility for packages and subsystems. It is probably simpler at the system (image file) level, as with Laurel.

To the world at large

If Lisp is chosen: It probably won't happen, except for a few research sites whose interests include both Lisp and high-capability personal computing.

If Mesa is chosen: It definitely won't happen.

Effect on Image

In the rest of Xerox

If Lisp is chosen: Software development groups throughout the corporation are likely to perceive this as a CSL "abandonment" of Mesa just as it is being adopted elsewhere. This may tend to lessen our credibility, and somewhat reduce the morale of Mesa proponents and users.

If Mesa is chosen: CSL would probably be perceived as doing research of greater relevance to the corporation. There would be a greater chance of development organizations converting to environments similar to EPE as its advantages are demonstrated (and as more capable hardware becomes cheaper).

In the world at large

Issue: Either choice will increase the "black hole" perception in some communities relevant to us.

If Lisp is chosen: The programming language, system implementation, and structured programming communities will find it harder to communicate and cooperate with us.

If Mesa is chosen: The AI, automatic programming, and theorem-proving communities will find it harder to communicate and cooperate with us.

Inter-Office Memorandum

To	Cedar Language Committee	Date	April 12, 1979
From	Bobrow & Deutsch	Location	Palo Alto
Subject	Another proposal for "objects" in Mesa	Organization	CSL

XEROX

Filed on: [Ivy]<CedarDocs>lang>BD-objects.memo

This memo describes a proposal for "objects" in Mesa, based on the current Mesa ideas of interface and module instance. It differs significantly from Peter Bishop's proposal, which is based on variant records. Our proposal requires few language and system changes and has withstood a certain amount of examination (by Lampson, Satterthwaite, and Morris); that does not mean, of course, that it definitely contains no lurking problems.

> Historical footnote: this proposal is quite close to the way instances, modules, and global frames were supposed to work in the original conception of Mesa in 1971....

Discussion

The current Mesa interface / IMPORT / EXPORT machinery already provides a notion of an abstract interface and concrete implementers. However, it suffers from some serious problems. Importers can only be linked to instances of implementers in two ways: at bind time, or through POINTER TO FRAME. The former mechanism requires a link in the importer's frame for each procedure used, and new instances cannot be created at run time or passed around; the latter mechanism requires the client to know the identity of the implementer.

The Binder already has a concept of an "interface record", which holds a vector of procedure descriptors (<global frame index, entry point number> pairs) as well as pointers to interface variables and signals. If interface records actually existed at run time, we would have solved half the problem, in that the interface record could serve as the representation of an object and a pointer to it could be passed around freely. This is a very slight extension to the present language -- it requires little more than replacing the word DEFINITIONS with the word RECORD! Unfortunately, it still leaves us with the object being represented by the ensemble of the interface record and the global frames of the procedures it contains. Now if we require that all the interface elements be exported by a single module, the global frame indices in the interface record become redundant and can be represented by a single index or pointer: in effect, the interface record is now just a map from operation numbers to entry points. If, finally, we arrange that the entry points of the implementers are in the same order as the declarations of the procedures in the interface, even this map becomes redundant, and all we need for the interface record is the global frame of the implementer.

The POINTER TO FRAME model is a better one for objects, since it only requires one copy of the procedure vector per implementation (in fact, the procedure vector is implicit in the entry point vector of the implementing program module), and only one data structure to represent an object. In this approach, our task is to find a way to allow a POINTER TO FRAME to point to any of a number of different frames, and yet have the client be able to refer to the exported objects identically. We can accomplish this by making sure the compiler allocates the exported objects identically in all implementers. This is the same conclusion that we reached starting from the other direction, and forms the basis of the proposal below.

Proposal

We define a new kind of DEFINITIONS module introduced by the keywords OBJECT DEFINITIONS. The semantics of such a module differ from those of an ordinary DEFINITIONS module only in the following ways:

If ObjDefs is an OBJECT DEFINITIONS module which is known to module ObjClient, then the type POINTER TO FRAME[ObjDefs] is available in ObjClient. Such a pointer may point to any global frame of any implementer of ObjDefs.

If ObjClient IMPORTs an instance of ObjDefs named ObjInst, the effect is exactly the same as defining ObjInst as a variable of type POINTER TO FRAME[ObjDefs]. However, in contrast to the usual behavior for IMPORTed interfaces, there is no way for the Binder or Loader to create an instance to store into ObjInst, since there is no way to tell which implementation of ObjDefs is wanted. (Note that if ObjInst = ObjDefs, i.e. the IMPORTS statement does not provide an explicit name, then ObjClient cannot refer to the interface ObjDefs at all, only to the instance pointer. Perhaps this should be an error.)

The only reason for allowing IMPORTing of OBJECT DEFINITIONS at all is to allow clients to write code that will work regardless of whether the interface is defined by DEFINITIONS or OBJECT DEFINITIONS. Note that in the current language, the only way to create appropriate instances for IMPORTed OBJECT DEFINITIONS is to have another module that IMPORTs ObjClient directly, so it can store into the ObjInst variables.

Part of the problem here is that the Binder cannot actually run the START code of the implementer, or supply parameters to it. If the Binder had those abilities, then it would be possible with little or no additional extension to C/Mesa to specify which implementer to instantiate for an IMPORTed object definition.

If ObjImpl is a program module that EXPORTs ObjDefs, it must export *all* the PROCEDUREs, SIGNALs, and variables defined as interface elements in ObjDefs, and it must not EXPORT any other interface. Furthermore, the compiler arranges the global frame of ObjImpl so that the interface elements are allocated at locations computable from ObjDefs alone.

The variable SELF is automatically defined in ObjImpl: its type is POINTER TO FRAME[ObjImpl] and it represents the frame of the currently executing instance of ObjImpl. This allows an implementer to get hold of its instance to pass as an argument, for example.

If ObjPtr is of type POINTER TO FRAME[ObjDefs], then for any interface element X in ObjDefs, ObjPtr.X is legal and is interpreted as though it were ImplPtr.X, where ImplPtr is of type POINTER TO FRAME[ObjImpl]. It does not matter which implementation ObjPtr actually points to, since all implementers of ObjDefs have their interface elements (global variables and signal codes, and entry points) laid out in the same way. In fact, it is legal to assign a POINTER TO FRAME[ObjImpl] to a POINTER TO FRAME[ObjDefs].

Creating new instances of objects is accomplished with a CREATE expression that is essentially equivalent to the current NEW and START constructs. CREATE ObjImpl[parameters] returns a POINTER TO FRAME[ObjImpl] just as NEW and START would; this is then usable as a POINTER TO FRAME[ObjDefs].

The difference between CREATE and NEW + START is that CREATE does not enter the frame into the global frame table (GFT). As a result, the usual representation for procedures in ObjImpl does not work. This is a problem only if ObjImpl actually needs to refer to one of its own procedures in some way other than a local call (e.g. it needs to pass it as a parameter). The

preferred solution to this problem is the one discussed under LONG POINTERs below. However, another solution is possible: if all global frames (objects) stay within the MDS, then it is possible to use the "indirect" form of procedure descriptor to obtain the desired effect, in a way similar to that used for inner procedures at present. The idea is to set up a two-word structure in the global frame, properly quadword-aligned, which contains the entry point number and a pointer back to the beginning of the frame, and to fabricate an indirect procedure descriptor that addresses this structure.

LONG POINTERs

It would be very nice if objects did not need to live within the same MDS, and could be designated by LONG POINTERs. Conceptually there is no problem with a LONG POINTER TO FRAME. Of course the system procedures that manipulate frames (like NEW and START) would have to be changed to take LONG POINTERs. Also, some changes would be entailed in the instruction set / virtual machine specification: G is not a LONG POINTER at present, and procedure invocations through POINTER TO FRAME are handled by actually constructing a procedure descriptor, which is limited to 16 bits. The main difficulty is that when an implementer really needs to construct a procedure descriptor for one of its own procedures, there is no way to pack the necessary information (LONG POINTER to SELF, and entry point number) into 16 bits, even using indirection.

One possible solution to this problem is to define a new LONG PROCEDURE type consisting of a LONG POINTER to a global frame and an entry point number. The bit representation would be arranged in such a way that one could inspect a cell and tell whether it was a PROCEDURE or the first word of a LONG PROCEDURE, so that the Call instructions which address the local or global frame would work equally well on ordinary and LONG descriptors. An instruction to lengthen a PROCEDURE would be very simple, similar to the Lengthen Pointer instruction, and would only need to be invoked when a procedure was needed in a non-call context. We advocate this solution for its relative simplicity and because it also solves another problem, namely the restriction on the number of global frames and the requirement that all global frames be accessible through an indexed table.

Known weaknesses in the proposal

Since an object is a global frame, it contains links for all procedures imported by its implementer. This is wasteful since every instance of the implementer (object) will presumably be bound the same. There are two ways to deal with this without further changes:

> Use the LINKS : CODE feature of the Binder, which moves all these links into the implementer's code segment.
>
> As much as possible, have implementers use objects for imported services rather than individual procedures. This reduces the overhead to one POINTER per object, rather than one PROCEDURE per procedure.

In this proposal, it is not possible for one module to implement more than one object type (except for a chain of subtypes), because the entry point (operation) indices for every object type are assumed to start at 1.

In this proposal it is also not possible for several modules to cooperatively implement an object type, because there is only one global frame per object. A solution to this problem is possible along the following lines:

> Let one of the implementer modules be the "principal" and the others be "assistants". Arrange things so that the global frame of the principal points to global frames for the assistants in a known way, and vice versa. In the entry vector of the principal, mark entries which are actually implemented in the assistants in a special way that also contains a pair

<offset in principal's global frame of pointer to assistant's global frame, assistant's entry point number>. Within an assistant, let SELF refer to the back pointer to the principal's global frame.

This solution is workable, although not very inspiring. It also requires that all exported objects other than procedures be exported by the principal, since the proposed scheme for automatic indirection of the entry point does not work for variables or signals.

The overhead for a global frame is currently 4 words: a LONG POINTER to the code segment, a size code for the frame, some flag bits, and the GFI (index in GFT). Attempting to eliminate one of the overhead words, which might be possible since objects do not need a GFI, would probably be inconvenient since the instruction set and runtime system assume the present format, and would have to check at access time.

Subclasses

Subclasses can be accommodated in this proposal by the principal / assistant mechanism. The subclass must be the principal, and the superclass must be the assistant. If OBJECT DEFINITIONS module SubDefs declared itself as a subclass of SuperDefs, the compiler could do the appropriate thing if a client tried to use a POINTER TO FRAME[SubDefs] as a POINTER TO FRAME[SuperDefs], namely follow the pointer from SubImpl's global frame to SuperImpl's. Note that this approach does work for multiple superclasses, and any decisions about conflicting interface element names can be handled at compile time. Note also that an automatic indirection mechanism is not required, since the subclass / superclass structure is known at compile time and the compiler can insert the extra levels of pointer-following in the code.

Inter-Office Memorandum

To	Cedar Language Committee	Date	March 2, 1979
From	Transcribed by Deutsch	Location	Palo Alto
Subject	Messages on Lampson's proposal for objects	Organization	CSL

XEROX

Filed on: [Ivy]<CedarDocs>lang>bwl-objects.memo

This file contains an exchange of messages on the subject of a proposal for "objects" in Mesa originated by Butler. There are two other proposals (by Howard Sturgis and by Peter Bishop) outstanding in this area; copies of these proposals will also appear on [Ivy]<CedarDocs>lang> in the near future.

From Lampson, 11/22/78

While discussing the inner meaning of Euclid modules with Jim Horning this morning, I was stimulated to develop the following idea for adding a very simple and conservative form of "objects" to Mesa. The idea is based on the observation that Euclid modules (which are a form of object, though not one which permits multiple implementations) can be regarded as sugar for a record declaration containing all the constants and variables of the module, together with the type and routine declarations of the module allowed to float free (at the same lexical level as the module itself). In addition, m.p(args) must be converted to p(mr, args), and perhaps some renaming is needed. Finally, Euclid modules enforce protection which allows a module invariant to be guaranteed, and this of course is lost.

The idea is this. Suppose x has type T (or is dereferenceable to type T), and T is declared in FooDefs. Then x.P[args] is rewritten to FooDefs.P[x, args], unless x itself has a procedure or array component named P. According to the manual, p 43, any necessary automatic dereferencing will be applied to x in the argument position (though I don't understand why that isn't also true for the right-sides of assignments).

This seems easy to understand and implement. It allows the programmer to treat T's just like Euclid modules, i.e. as honest objects, though without the possibility of multiple implementations. Mesa already provides the protection which Euclid modules provide. As far as I can see, there are no conflicts with other, more wonderful flavors of objects or uniform reference which we might want to introduce later.

Comments are welcome.

From Deutsch, 11/22/78

There are a couple of things I question about your proposal.

What if FooDefs is imported into the current module more than once? I suppose x.P will get bound to the instance (assuming there is one) that is named FooDefs, the same as the module name.

A consequence of this proposal is that you can't have more than one 'object' type with the same operation name declared in an interface. This is because you're disambiguating the name at the

interface level, rather than the type level.

Peter Bishop's proposal handled the possibility that the object itself (or the concrete type) defined the procedures. While you aren't dealing with this in your present proposal, I do think he was right in that the question of INHERITANCE (of components, in this case operations) from the type or interface to the object should be separated from the question of 'OBJECTIFYING' an operation by automatically supplying to a procedure the same object that was used to turn the operation name into a procedure.

I haven't been able to come up with a good counter-proposal. The lines I'm thinking along are to allow constant fields in records (we really need them anyway), including 'quasi-constant' procedures that simply make x.Foo a synonym (in effect) for Defs.Fie, and a notation more like Bishop's for saying that an operation needs to be 'objectified'.

I don't think there is any harm in Butler's proposal, and it doesn't seem to interfere with later additions, but I think there is some chance that it will not be compatible with them in the sense that a more general solution may require some definitions to be rewritten.

From Satterthwaite, 11/24/78

Your proposal sounds interesting, and it prodded me into looking a bit at a "least-cost" implementation. That raised some further issues.

First, a clarification on automatic dereferencing. The idea is that the '[' operator forces dereferencing of its left operand; i.e., p[x] -> p↑[x]. This was intended to make the use of POINTER TO ARRAY ... more pleasant. It also works for POINTER TO PROCEDURE ... . In no case, is there automatic dereferencing of x, which is indeed treated as an ordinary right side. The cited statement in the manual (p. 43) certainly is easy to misinterpret and should be clarified (as in the fine point on the same page).

[In the current language, there is no fundamental problem with automatic dereferencing of right sides. As I recall, we rejected it partly by popular demand and partly because we weren't sure what ugly interactions, if any, there might be with uniform references, schemes, or other ideas for the grand and glorious future.]

To your proposal. Most of the following comments can be read at two levels. At one level, they raise some questions about the "right" semantics; at the second, they just address your remark that the proposal is "easy to ... implement", i.e., some complications or restictions would arise from an implementation that tries to minimize changes to the current compiler.

(1) Deciding where T is "declared" is not entirely obvious. Consider:

D1: DEFINITIONS = ... T: TYPE = D0.T'; ...

D0: DEFINITIONS = ... T': TYPE = RECORD [...]; ...

This situation is fairly common; T' is redefined either to make things more convenient for users of D1, to propagate types "upward" in some heirarchy, or to reflect some methodology for the division and distribution of interfaces (e.g., Pilot's conventions).

As the compiler is currently structured, T appears to be declared in D0, at least in the following sense: given x: POINTER TO D1.T and x.P[args], the compiler will have in hand a structure pointing into D0 when it discovers that, after applying all suitable coercions, P is not a valid field name. It would therefore be prepared to look for P in D0 but not in D1. Consider the following alternatives:

D1: DEFINITIONS = ... T: TYPE = PRIVATE D0.T';

D1: DEFINITIONS = ... T: TYPE = RECORD [val: D0.T'];

D1: DEFINITIONS = ... T: TYPE = RECORD [val: PRIVATE D0.T'].

The third of these is pretty sure to make it look like T is declared in D1 (at least for clients; see point 3). The first probably should also (but requires a bug fix in the compiler). The second raises questions about single component records, the next point.

(2) Mesa tries to get a lot of work out of the record concept (perhaps beyond all reason). Because procedures always return records, we have the rule that record brackets can be stripped automatically from all single component records. On the other hand, encapsulation within records is Mesa's "painting" mechanism, and painting often produces a single-component record type. These two uses do not always interact very well. Your proposal seems to make the situation somewhat worse. In trying to interpret x.P[args], how soon should the compiler decide that P is not a valid field selector?. Given the second redefinition of T above, the current compiler would look in the fields of T for P; failing that, it would look at the fields of T' (and be left pointing into D0 again when that failed).

(3) Even the first and third forms give anomalous answers to the question of where T is declared for an implementer of D1 (which has "shared" access to that interface); for such an implementer, "PRIVATE" essentially disappears and we are back to point (1) or (2).

(4) [Expanding on one of Peter's points]. If a client imports several instances of FooDefs, things are a little confusing. A related issue has already arisen in Mesa 5. It is possible to specify, in the type declaration, default values for the fields of a record (if time and space permit, for any named type at all). Mesa 5 also introduces variables (so-called "imported variables") as interface components. These were intended primarily for use by inline procedures, but they give a sensible interpretation to non-constant defaults specified in a definitions module. Given

D: DEFINITIONS = ... x: INTEGER; -- implemented as a pointer to an integer var in the exporter ... T: TYPE = RECORD [... f: INTEGER ← x, ...];

problems arise if several instances of D are imported. We can make this situation illegal (by, e.g., requiring constant defaults in TYPE declarations - the most likely short-term solution). If legal, then we must view such types as being associated with a particular instance of an interface, and the equivalence rules must be established. Our previous position was that the types were equivalent, but variables declared using the types got initialized differently. In the context of your proposal, we might want to reconsider. If x and y come from different instances of an interface, then the assignment x ← y will change the set of operations applied to y (or y↑) without changing its representation - probably an invitation to disaster. On the other hand, if we keep the types separate, we get a limited form of multiple implementation (though the implementers must work on the same object types and use, e.g., loopholes or useless discriminations for now) -- perhaps so limited as to be useless.

(5) [A minor point]. In the rewriting of x.P[args] as FooDefs.P[x, args], what are the exact rules? Presumably, we undo all the coercions involved in discovering that P was not a legitimate field name and start over on the original x (with a new set of rules). For adding x to the argument list, do we use a positional convention (always the first argument), a distinguished keyword ("SELF"?), a keyword established in the definitions module (as in monitors)?

I guess I feel that all these problems can be resolved in a reasonable way (or given a least-effort solution adequate for the purposes of some initial experimentation). The source code in clients seems to have a good chance of surviving transitions to a more elaborate approach to objects. I agree with Peter that later rewriting of definitions (and perhaps implementers) is likely, but that doesn't seem so terrible if we really want to encourage the object style (and I think we do). That brings me to

(6) We believe that a more general solution is necessary for the long term. Can/should we do in the relatively short term (when this proposal is most interesting)? Quite properly, there is no mechanism for introducing experimental language features (that have high probability of being supplanted or withdrawn) into standard releases by SDD. On the other hand, there is currently no condoned way of producing a compiler that is not part of a standard release but is recognized for purposes of support and maintenance.

From Morris, 11/27/78

It seems quite reasonable, but doesn't do much except save a little typing. There must be more to objects than this; or is there? I seems that what is still missing is the ability to vary the implementation at run time. Jim

From Lampson, 11/27/78

Peter's message:

About multiple importing: I think you are right - that is certainly the most straightforward rule. It might be possible to keep track of the version which was used when x was declared, but I am not sure that the compiler has this information, or that it would really be a good idea to use it in any case.

Your comment about reusing the operation name is certainly right. A more grandiose proposal would also use the type of the object to select the procedure, but this would have more far-reaching consequences for the existing Mesa system, since there would now be two entities in the same scope with the same name. In the long run, I'm sure that is what is wanted, but it needs to be worked out much more carefully. The bad consequence of this is that one might be forced to give different names to oeprations which would more naturally have the same name.

Ed's message:

Automatic dereferencing: Rephrasing the manual would be good - even after reading your message, I had to read it twice to see how I had gone wrong, though it sounds fine now. Before having my mind polluted by the correct interpretation, I didn't find that the fine point clarified things at all. I think automatic dereferencing of x in its rightside context is really what is needed for my proposal, but I am not sure I want it in general. Perhaps no automatic dereferencing would do for a start.

(1-3) The fundamental issue here seems to be whether we want to take the view that all operations of an object should be defined in the same interface, together with the object, or that new operations can be added fairly freely. I incline toward the former, and from that position claim that looking in D0 is the right thing to do. Are there examples from current practice which contradict this claim. The other, more anarchic position would require starting over and looking first at D1, then at D0. This seems harder to implement and to explain, and the advantages are unclear to me now. I think the conservative position gets rid of the problems about single-component records and PRIVATE?

(4) All this confirms my earlier position that the simplest interpretation is best for now.

(5) Using the original x was what I had in mind. I guess that both positional (first-arg) and keyword (SELF seems fine) argument lists must be handled, since mixing is not allowed.

I would like opinions on the need to rewrite definitions and implementers. It seems to me that programs written to work under this proposal should continue to work, though they might not be the preferred way to do things in the glorious future. Do others agree?

From Satterthwaite, 11/27/78

Butler's message:

Automatic dereferencing: I don't see why your proposal requires automatic rightside dereferencing (as opposed to "dot" dereferencing); indeed, it seems more likely to require "referencing" (the procedure surely wants a pointer - that's the motivation for starting over again with x if it is a pointer originally). Am I missing something?

(1-3) Upon reflection, I decided I too preferred the more conservative position (that the record representing an object and all the operations be declared in the same interface). I guess this is easy for users to understand if they look upon the procedures in that interface as (potential) extra fields that have been moved out of the record. This position seems to minimize implementation problems and anomolies.

On adding new operations: this is probably a step in the wrong direction (building on kludges?), but I point out another observation for what it's worth. Refer back to my example. If

D1: DEFINITIONS = ... T: TYPE = RECORD [val: D0.T']

it would be easy to check D1 for procedure P before stripping the record and going to D0. In this limited sense, T and the operations in D1 "extend" T' and the operations in D0. Also, T' ← T is automatic, but not vice-versa.

(4) [also your reply to Peter's message]

It's true that the compiler currently doesn't have the information about which instance of an interface a type comes from. For reasons outlined in my previous message, I have already been looking into this problem. It appears that the bookkeeping changes are easy to arrange. If we accept this, however, it becomes hard to justify preserving any distinction between "included" and "imported" modules; the IMPORTS clause replaces the DIRECTORY, etc. We have discussed these changes before; they seem to make sense in the long run but cause some compatibility problems.

If the conservative view is adopted (so that single-component PRIVATE records do not cause anomalies), I believe that the sources of clients and implementers of a class have a good chance of surviving a change to a better object scheme, even for that class (if, in the case of implementers, a dynamically varying implementation is not needed). Obviously, the definitions won't have this property. I guess your scheme could continue to be supported in the glorious future; I am more worried about how it would interact with generic procedures, uniform references, etc., than with objects a la Peter Bishop.

From Lampson, 11/27/78

The reason for wanting automatic dereferencing is this. Suppose I write x.P[y], and x turns out to be POINTER TO T, where P is in T's def module Tdefs, and is declared as PROC [xx: T, ...]. Then x.P[y] will be converted into Tdefs.P[x, y] and will then result in a type error. What is needed, in the absence of automatic dereferencing, is Tdefs.P[x↑, y].

Your comment on adding new operations seems to be going in the opposite direction from your preference for the more conservative position that all operations should be defined in the same place. Was it intended just as a curiosity, or an attempt to be fair to the other side, or what?

As you know, I am all for getting rid of the DIRECTORY / IMPORTS distinction.

A slight modification to my proposal would be to require writing OBJECT PROCEDURE (or something like that) for procedure declarations in an interface which are to be eligible for this

wonderful transformation. This has the advantage of making the programmer's intentions clearer, and also of making it easier to automatically convert such source code into some other form in the glorious future.

From Horning, 3/2/79

Howard,

I have read your Nov. 16, 1976 memo, and agree that it sounds surprisingly moderate and "modern" in the context of Cedar. I found the first few sections to be rather heavy going, but think that your suggestions in sections 6 and 7 certainly bear re-evaluation.

Although many of the details differ (and it doesn't support multiple implementations), Butler's proposal of Nov. 22, 1977 has much the same flavor. I trust that somewhere in the region sketched out by the correspondence that triggered, your suggestions, and Ed's paper on "exported types" we can strike the right balance among generality, safety, ease of use, and efficiency of implementation.

Date: 14 Feb 1979 9:33 am (Wednesday)
From: Satterthwaite
Subject: Mesa 5
To: Cedar Language Features Design Committee: Bobrow, Deutsch, Horning,
Morris, Satterthwaite;
cc: Lampson

John Wick should be announcing a documentation package for alpha users later today. The language and compiler changes are documented in [Iris]<Satterthwaite>Compiler50.Bravo. In addition to the features described there, the data structures and most of the code now exist to support the following:

(1) Default initialization associated with any named type (not just record fields).

(2) "Objects", a la Lampson.

(3) "Exported" types, as described in the Mesa/Pilot paper that Hugh Lauer and I did.

(4) Interfaces parametrized with respect to other interfaces (so that one interface can "import" another), done some very simple way.

Some of these features might appear in Mesa 5 during alpha test. Items (2)-(4) are tips of icebergs, and the problem is more one of trying to remain compatible with future, more general solutions than of implementing something adequate to handle the simple cases.

Date: 14 Feb 1979 11:44 am (Wednesday)
From: Satterthwaite
Subject: Some Preliminary Thoughts
To: Cedar Language Features Design Committee: Bobrow, Deutsch, Horning,
Morris, Satterthwaite;
cc: Lampson

I have jotted down some initial thoughts about "safe language" and "objects" in [Ivy]<Satterthwaite>CedarThoughts.Memo. This is mostly an attempt to summarize and put on record ideas that have been kicking around in the Mesa Language Working Group for quite some time, and I'm sure that most of you have already seen various subsets of the proposals. Most are pretty conservative (but more or less compatible with the structure of the current compiler).

Date: 15 Feb 1979 2:55 pm (Thursday)
From: Satterthwaite
Subject: Mesa LWG Documents
To: Deutsch
cc: cedar-lang↑

I have gone through my files and assembled the collection of LWG minutes that appear to relevant on [Iris]<Satterthwaite>lwg-minutes.press (11 pages). I don't know how much sense the minutes will make out of context. They include discussions of the following design documents:

[Iris]<Wick>LWG>Sequences.Bravo
[Iris]<Satterthwaite>InlineProcedures.Memo
[Iris]<Satterthwaite>DefaultFields.Memo
[Iris]<Satterthwaite>VarLength.Memo (never fleshed out completely)

The current Mesa task list (not updated to reflect Mesa 5) is on

[Iris]<Wick>Planning>MesaTasks50.Bravo

There are other documents on [Iris]<Sweet>LWG> and [Iris]<Wick>LWG> (including some older minutes of the LWG), but they seem to be obsolete and/or only slightly relevant (design of processes, etc.). Note that the task list has a very long list of references; I can probably find a hardcopy of any that appear relevant.

Date: 16 FEB 1979 1404-PST
From: DEUTSCH
Subject: Mesa language design documents
To: bobrow, horning, morris, satterthwaite

On [Ivy]<CedarDocs>language>mlwg>*.press you will find a set of memos from the Mesa Language Working Group that Ed Satterthwaite suggested would be useful to us. You can print them directly, or, if you are feeling paper-conscious, you can make double-sided photocopies from the copy I printed. (If you prefer the latter, let me know and I'll have Dianne do it.) ------

Date: 18 Feb 1979 4:06 pm (Sunday)
From: Satterthwaite
Subject: More References
To: cedar-lang↑
cc: Lampson

I'm sure that everyone has plenty to read, but here's the missing reference from my "preliminary thoughts" message:

> Data Types as Values: Polymorphism, Type-checking, Encapsulation
> A. Demers, J. Donahue, G. Skinner
> 5th POPL Proceedings (Jan 1978)
> 23-30.

Jim Donahue also talked about this paper when he was here interviewing. I rather liked the paper and the ideas behind it. On the other hand, I was first exposed to it under possibly judgement-warping conditions, with nothing more promising to read while sitting on airplanes between Ithaca and San Jose last winter.

It's probably also worth looking at Peter Bishop's stuff on objects and such - Peter put a lot of thought and work into his proposals and we ought to be able to benefit from that somehow (even if only by knowing a direction of development to avoid).

Date: 21 Feb 1979 5:29 pm (Wednesday)
From: Horning
Subject: Safe Language Subcommittee
To: Sturgis
cc: Satterthwaite, Cedar-Lang↑, Lampson

Howard,

Just an expansion and confirmation of our corridor conversation:

The Safe Language Subcommittee will have its first meeting next Monday at 10:30; I understand that your prior commitment to the eclipse cannot be shifted, but Ed and I would welcome your written comments prior to the meeting.

We do not expect to make any binding decisions at this meeting. Our charter is to produce a working paper for the Language Features Committee that catalogs the issues to be dealt with, and provides at least tentative orderings in terms of 1) dependencies, 2) time when needed within Cedar, and 3) importance for Cedar.

We have identified at least two aspiration levels of "safety" for Cedar: -consistent type interpretation of storage, and -possibility of safe automatic deallocation (no dangling references). We will probably focus on the latter (stronger) level, since deallocation is one of the major motivations for defining a safe subset in any case.

Our catalog of issues currently contains the following: -Identify necessary run-time safety enforcement mechanisms, e.g.:
-array bounds checking,
-forced initialization of pointer variables. -Define a definitely safe subset of "checked" Mesa. -Provide a linguistic mechanism for making the safety boundary explicit in programs. -Provide substitutes for enough common special cases of safe features to allow "95%" of CedarMesa modules to be written in the safe language. Examples:
-VAR parameters,
-strengthened relative pointers. -Verify that automatic deallocation is feasible and safe given the above changes.

I'm sure that you can add a thing or two to this catalog. We'd also be interested in your personal list of kinds of "necessary loopholes."

Jim H.

Date: 25 Feb 1979 11:22 pm (Sunday)
From: Satterthwaite
Subject: Exported Types
To: lwg↑, cedar-lang↑, mesaimplementers↑, Lauer, Redell, McJones, Levin, Sturgis

A number of people have expressed interest in the idea of "exported types" as introduced in the paper by Hugh Lauer and myself. (The idea that something like this should be possible goes back to some conversations with Jim Morris several years ago.) I have been exploring the idea and have uncovered some surprises. My discoveries to date are summarized by the memo

<Satterthwaite>ExportedTypes.Memo/.Press (on [Iris] and [Ivy]).

I am not sure what to make of all this and would appreciate your comments and advice.

Ed

Date: 24 Feb 1979 4:42 pm (Saturday)
From: Morris
Subject: Some proposals
To: Cedar-lang↑, Scott

I have written a rough memo on some moderately modest proposals for Cedar/Mesa. See [ivy]<cedardocs>lang>proposals.memo.

I'll see you all in a few weeks.

Jim

Date: 27 Feb 1979 11:25 am (Tuesday)
From: Horning
Subject: Unsafe at any speed?
To: CSL↑, PE-Interest↑

This message solicits input about the unsafe features of Mesa, and the known safe uses to which these features are put.

One of the language capabilities to be provided in Cedar Mesa is automatic storage deallocation (via garbage collection or reference counting) for at least a subset of a program's dynamically allocated storage. Several features in the present Mesa language make it infeasible to do such automatic deallocation safely. We are currently starting the design of a "safe subset" of the language to cope with this problem. To first approximation, the safe subset will be that part of the language where even incorrect programs cannot interfere with the operation of the automatic deallocator.

Before presenting our current catalog of unsafe features, some context is appropriate:

-We do NOT propose to ban the unsafe features entirely from Cedar Mesa, merely to restrict or prohibit their use within safe regions.

-There will be some textual means of indicating that a region of a program (e.g., a module or a procedure) is UNSAFE. This will inhibit compiler enforcement of the safety restrictions; conversely, it indicates a region where the programmer has assumed additional responsibility for the correctness and safety of his program.

-It is our intention that the vast majority of Cedar modules will consist entirely of safe regions. Unsafe regions should be the exception, and introduced only for sufficient reason. However, we do not think that it is worth the effort at this time to introduce enough language constructs bring the last 5% of low-level modules into the safe language.

-The following list is very preliminary, and circulated now primarily to elicit further information and comment. IT IS NOT A PROPOSAL, but merely a catalog of known issues.

LANGUAGE FEATURES KNOWN TO BE UNSAFE, BUT FREQUENTLY USED IN SAFE WAYS

[We would like to capture most of the safe uses in more restricted language features.]

-The "address of" operator, @. [Issue: should we add VAR parameters to the language?]

-Procedure values. [Issues: Can we restrict these to global procedures? Should we introduce "persistent closures" to make these safe?]

-Variant records. [Problems: change of access path or variant within a discriminating SELECT; computed and overlaid variants. Issues: adopt Euclid rules for discrimination? Disallow computed and overlaid variants when PC (pointer or procedure containing) fields are involved?]

-Creation and modification of descriptors. [Issue: Can a sufficiently safe ALLOCATE statement be devised?]

-IO, especially disk IO. [Issue: Should the type system extend to the file system? Can types be made sufficiently "persistent" that this will never make files inaccessible?]

INHERENTLY UNSAFE FEATURES THAT MAY HAVE TO BE BANNED FROM THE SAFE LANGUAGE.

-LOOPHOLE into a PC type. [Issue: The "skeleton type system" permits a weaker restriction, but is it worth it?]

-Uninitialized PC or subrange variables. [Solve by having the compiler force initialization whenever the programmer does not supply an explicit initialization.]

-Subscript out of bounds, dereference of NIL. [Supplement compile-time knowledge with run-time checks as necessary.]

-UNSPECIFIED.

-Relocation schemes that rely on non-interruptability of expressions.

-Machine-coded procedures. [Some standard procedures (e.g., bit operations) are safe, and their call could be allowed in safe regions.]

-Pointer arithmetic. [The safe language will have "stronger" relative pointers than current Mesa, however.]

-Explicit freeing of PC storage.

-Destroying a global frame.

QUESTIONS

-Do these two lists together cover the unsafe language features of which you are aware?

-Are there common safe uses of features on the second list that could be captured in safe language features?

-What else should we consider about your usage of unsafe features?

Jim H.

Date: 27 Feb 1979 1:48 pm (Tuesday)
From: Taft
Subject: Re: Unsafe at any speed?
In-reply-to: Your message of 27 Feb 1979 11:25 am (Tuesday).
To: Horning
cc: Taft

This may seem obvious, but... I recall that during past discussions, whenever anyone mentioned automatic storage allocation, invariably someone else would pipe up with "you mean automatic storage DEallocation". The asymmetry implied by this sometimes bothered me. It seems to me that the logical counterpart of automatic storage deallocation is automatic storage allocation, by which I mean you allocate objects, not uninterpreted blocks of memory. If Cedar provided a mechanism by means of which a type could be instantiated, then the need for a lot of Mesa's existing unsafe facilities would be reduced or eliminated. (I imagine this operator would look something like a constructor, but one that allocated storage as well as laying down bits.)
Ed

Date: 27 Feb 1979 1:59 pm (Tuesday)
From: Taft
Subject: p.s.
To: Horning

Have you ever used either ECL or PPL? They both provide the sort of automatic storage allocation I am thinking about. But, then, so does Smalltalk, though somehow it feels different.
Ed

Date: 27 Feb 1979 3:25 pm (Tuesday)

From: Horning
Subject: Re: Unsafe at any speed?
In-reply-to: Your message of 27 Feb 1979 1:48 pm (Tuesday).
To: Taft
cc: Horning

Ed,

Mesa has one such mechanism: module instantiation. Unfortunately, Alto Mesa makes this expensive and limits you to a total of 256 such instances. We're going to try to do better in Cedar. However, the remark still stands: automatic allocation is easy--it's the deallocation that's hard to do automatically.

Jim H.

Date: 2 Mar 1979 3:57 pm (Friday)
From: Horning
Subject: "Mesa conventions for PreMesa compatible objects"
To: Sturgis
cc: Lampson, Deutsch, Satterthwaite, Bobrow, Morris, Horning

Howard,

I have read your Nov. 16, 1976 memo, and agree that it sounds surprisingly moderate and "modern" in the context of Cedar. I found the first few sections to be rather heavy going, but think that your suggestions in sections 6 and 7 certainly bear re-evaluation.

Although many of the details differ (and it doesn't support multiple implementations), Butler's proposal of Nov. 22, 1977 has much the same flavor. I trust that somewhere in the region sketched out by the correspondence that triggered, your suggestions, and Ed's paper on "exported types" we can strike the right balance among generality, safety, ease of use, and efficiency of implementation.

Date: 22 Nov 1978 3:04 pm (Wednesday) From: Lampson Subject: A modest proposal for objects in Mesa To: Horning, Mitchell, Geschke, Levin, Morris, Deutsch, Satterthwaite, Wick, Johnsson, Sweet cc: Lampson

While discussing the inner meaning of Euclid modules with Jim Horning this morning, I was stimulated to develop the following idea for adding a very simple and conservative form of "objects" to Mesa. The idea is based on the observation that Euclid modules (which are a form of object, though not one which permits multiple implementations) can be regarded as sugar for a record declaration containing all the constants and variables of the module, together with the type and routine declarations of the module allowed to float free (at the same lexical level as the module itself). In addition, m.p(args) must be converted to p(mr, args), and perhaps some renaming is needed. Finally, Euclid modules enforce protection which allows a module invariant to be guaranteed, and this of course is lost.

The idea is this. Suppose x has type T (or is dereferenceable to type T), and T is declared in FooDefs. Then x.P[args] is rewritten to FooDefs.P[x, args], unless x itself has a procedure or array component named P. According to the manual, p 43, any necessary automatic dereferencing will be applied to x in the argument position (though I don't understand why that isn't also true for the right-sides of assignments).

This seems easy to understand and implement. It allows the programmer to treat Ts just like Euclid modules, i.e. as honest objects, though without the possibility of multiple implementations. Mesa already provides the protection which Euclid modules provide. As far as I can see, there are no conflicts with other, more wonderful flavors of objects or uniform reference which we might want to introduce later.

Comments are welcome. ---

Date: 22 Nov 1978 3:13 pm (Wednesday) From: Horning Subject: Re: A modest proposal for objects in Mesa In-reply-to: Your message of 22 Nov 1978 3:04 pm (Wednesday). To: Lampson cc: Horning, Mitchell, Geschke, Levin, Morris, Deutsch, Satterthwaite, Wick, Johnsson, Sweet

Butler,

Your golden pen has convinced me again. I'll probably regret it when I sober up, though!

Jim H.

Date: 22 NOV 1978 1712-PST From: DEUTSCH Subject: Re: A modest proposal for objects in Mesa To: Lampson, Horning, Mitchell, Geschke, Levin, Morris, To: Satterthwaite, Wick cc: DEUTSCH

In response to the message sent 22 Nov 1978 3:04 pm (Wednesday) from Lampson

There are a couple of things I question about your proposal.

What if FooDefs is imported into the current module more than once? I suppose x.P will get bound to the instance (assuming there is one) that is named FooDefs, the same as the module name.

A consequence of this proposal is that you can't have more than one 'object' type with the same operation name declared in an interface. This is because you're disambiguating the name at the interface level, rather than the type level.

Peter Bishop's proposal handled the possibility that the object itself (or the concrete type) defined the procedures. While you aren't dealing with this in your present proposal, I do think he was right in that the question of INHERITANCE (of components, in this case operations) from the type or interface to the object should be separated from the question of 'OBJECTIFYING' an operation by automatically supplying to a procedure the same object that was used to turn the operation name into a procedure.

I haven't been able to come up with a good counter-proposal. The lines I'm thinking along are to allow constant fields in records (we really need them anyway), including 'quasi-constant' procedures that simply make x.Foo a synonym (in effect) for Defs.Fie, and a notation more like Bishop's for saying that an operation needs to be 'objectified'.

I don't think there is any harm in Butler's proposal, and it doesn't seem to interfere with later additions, but I think there is some chance that it will not be compatible with them in the sense that a more general solution may require some definitions to be rewritten. ------ ---

Date: 24 Nov 1978 11:21 am (Friday) From: Satterthwaite Subject: Re: A modest proposal for objects in Mesa In-reply-to: Your message of 22 Nov 1978 3:04 pm (Wednesday). To: Lampson cc: Horning, Mitchell, Geschke, Levin, Morris, Deutsch, Satterthwaite, Wick,
Johnsson, Sweet

Your proposal sounds interesting, and it prodded me into looking a bit at a "least-cost" implementation. That raised some further issues.

First, a clarification on automatic dereferencing. The idea is that the '[' operator forces dereferencing of its left operand; i.e., p[x] -> p↑[x]. This was intended to make the use of POINTER TO ARRAY ... more pleasant. It also works for POINTER TO PROCEDURE ... . In no case, is there automatic dereferencing of x, which is indeed treated as an ordinary right side. The cited statement in the manual (p. 43) certainly is easy to misinterpret and should be clarified (as in the fine point on the same page).

[In the current language, there is no fundamental problem with automatic dereferencing of right sides. As I recall, we rejected it partly by popular demand and partly because we weren't sure what ugly interactions, if any, there might be with uniform references, schemes, or other ideas for the grand and glorious future.]

To your proposal. Most of the following comments can be read at two levels. At one level, they raise some questions about the "right" semantics; at the second, they just address your remark that the proposal is "easy to ... implement", i.e., some complications or restictions would arise from an implementation that tries to minimize changes to the current compiler.

(1) Deciding where T is "declared" is not entirely obvious. Consider:

D1: DEFINITIONS = ... T: TYPE = D0.T'; ...

D0: DEFINITIONS = ... T': TYPE = RECORD [...]; ...

This situation is fairly common; T' is redefined either to make things more convenient for users of D1, to propagate types "upward" in some heirarchy, or to reflect some methodology for the division and distribution of interfaces (e.g., Pilot's conventions).

As the compiler is currently structured, T' appears to be declared in D0, at least in the following sense: given x:

POINTER TO D1.T and x.P[args], the compiler will have in hand a structure pointing into D0 when it discovers that, after applying all suitable coercions, P is not a valid field name. It would therefore be prepared to look for P in D0 but not in D1. Consider the following alternatives:

```
D1: DEFINITIONS = ... T: TYPE = PRIVATE D0.T';

D1: DEFINITIONS = ... T: TYPE = RECORD [val: D0.T'];

D1: DEFINITIONS = ... T: TYPE = RECORD [val: PRIVATE D0.T'].
```

The third of these is pretty sure to make it look like T is declared in D1 (at least for clients; see point 3). The first probably should also (but requires a bug fix in the compiler). The second raises questions about single component records, the next point.

(2) Mesa tries to get a lot of work out of the record concept (perhaps beyond all reason). Because procedures always return records, we have the rule that record brackets can be stripped automatically from all single component records. On the other hand, encapsulation within records is Mesa's "painting" mechanism, and painting often produces a single-component record type. These two uses do not always interact very well. Your proposal seems to make the situation somewhat worse. In trying to interpret x.P[args], how soon should the compiler decide that P is not a valid field selector?. Given the second redefinition of T above, the current compiler would look in the fields of T for P; failing that, it would look at the fields of T' (and be left pointing into D0 again when that failed).

(3) Even the first and third forms give anomolous answers to the question of where T is declared for an implementer of D1 (which has "shared" access to that interface); for such an implementer, "PRIVATE" essentially disappears and we are back to point (1) or (2).

(4) [Expanding on one of Peter's points]. If a client imports several instances of FooDefs, things are a little confusing. A related issue has already arisen in Mesa 5. It is possible to specify, in the type declaration, default values for the fields of a record (if time and space permit, for any named type at all). Mesa 5 also introduces variables (so-called "imported variables") as interface components. These were intended primarily for use by inline procedures, but they give a sensible interpretation to non-constant defaults specified in a definitions module. Given

```
D: DEFINITIONS =
  ...
  x: INTEGER;          -- implemented as a pointer to an integer var in the exporter
  ...
  T: TYPE = RECORD [... f: INTEGER ← x, ...];
```

problems arise if several instances of D are imported. We can make this situation illegal (by, e.g., requiring constant defaults in TYPE declarations - the most likely short-term solution). If legal, then we must view such types as being associated with a particular instance of an interface, and the equivalence rules must be established. Our previous position was that the types were equivalent, but variables declared using the types got initialized differently. In the context of your proposal, we might want to reconsider. If x and y come from different instances of an interface, then the assignment x ← y will change the set of operations applied to y (or y↑) without changing its representation - probably an invitation to disaster. On the other hand, if we keep the types separate, we get a limited form of multiple implementation (though the implementers must work on the same object types and use, e.g., loopholes or useless discriminations for now) -- perhaps so limited as to be useless.

(5) [A minor point]. In the rewriting of x.P[args] as FooDefs.P[x, args], what are the exact rules? Presumably, we undo all the coercions involved in discovering that P was not a legitimate field name and start over on the original x (with a new set of rules). For adding x to the argument list, do we use a positional convention (always the first argument), a distinguished keyword ("SELF"?), a keyword established in the definitions module (as in monitors)?

I guess I feel that all these problems can be resolved in a reasonable way (or given a least-effort solution adequate for the purposes of some initial experimentation). The source code in clients seems to have a good chance of surviving transitions to a more elaborate approach to objects. I agree with Peter that later rewriting of definitions (and perhaps implementers) is likely, but that doesn't seem so terrible if we really want to encourage the object style (and I think we do). That brings me to

(6) We believe that a more general solution is necessary for the long term. Can/should we do in the relatively short term (when this proposal is most interesting)? Quite properly, there is no mechanism for introducing experimental language features (that have high probability of being supplanted or withdrawn) into standard releases by SDD. On the other hand, there is currently no condoned way of producing a compiler that is not part of a standard release but is recognized for purposes of support and maintenance.

Date: 27 Nov 1978 9:01 am (Monday) From: Morris Subject: Re: A modest proposal for objects in Mesa In-reply-to: Your message of 22 Nov 1978 3:04 pm (Wednesday). To: Lampson cc: Horning, Mitchell, Geschke, Levin, Morris, Deutsch, Satterthwaite, Wick,
Johnsson, Sweet

It seems quite reasaonable, but doesn't do much except save a little typing. There must be more to objects than this; or is there? I seems that what is still missing is the ability to vary the implementation at run time. Jim

---

Date: 27 Nov 1978 10:18 am (Monday) From: Lampson Subject: Re: A modest proposal for objects in Mesa In-reply-to: Your message of 22 NOV 1978 1712-PST. To: Deutsch, Satterthwaite cc: Lampson, Horning, Mitchell, Geschke, Levin, Morris, Wick

Peter's message:

About multiple importing: I think you are right - that is certainly the most straightforward rule. It might be possible to keep track of the version which was used when x was declared, but I am not sure that the compiler has this information, or that it would really be a good idea to use it in any case.

Your comment about reusing the operation name is certainly right. A more grandiose proposal would also use the type of the object to select the procedure, but this would have more far-reaching consequences for the existing Mesa system, since there would now be two entities in the same scope with the same name. In the long run, I'm sure that is what is wanted, but it needs to be worked out much more carefully. The bad consequence of this is that one might be forced to give different names to oeprations which would more naturally have the same name.

Ed's message:

Automatic dereferencing: Rephrasing the manual would be good - even after reading your message, I had to read it twice to see how I had gone wrong, though it sounds fine now. Before having my mind polluted by the correct interpretation, I didn't find that the fine point clarified things at all. I think automatic dereferencing of x in its rightside context is really what is needed for my proposal, but I am not sure I want it in general. Perhaps no automatic dereferencing would do for a start.

(1-3) The fundamental issue here seems to be whether we want to take the view that all operations of an object should be defined in the same interface, together with the object, or that new operations can be added fairly freely. I incline toward the former, and from that position claim that looking in D0 is the right thing to do. Are there examples from current practice which contradict this claim. The other, more anarchic position would require starting over and looking first at D1, then at D0. This seems harder to implement and to explain, and the advantages are unclear to me now. I think the conservative position gets rid of the problems about single-component records and PRIVATE?

(4) All this confirms my earlier position that the simplest interpretation is best for now.

(5) Using the original x was what I had in mind. I guess that both positional (first-arg) and keyword (SELF seems fine) argument lists must be handled, since mixing is not allowed.

I would like opinions on the need to rewrite definitions and implementers. It seems to me that programs written to work under this proposal should continue to work, though they might not be the preferred way to do things in the glorious future. Do others agree? ----------

Date: 27 Nov 1978 11:53 am (Monday) From: Satterthwaite Subject: Re: A modest proposal for objects in Mesa In-reply-to: Your message of 27 Nov 1978 10:18 am (Monday). To: Lampson cc: Deutsch, Satterthwaite, Horning, Mitchell, Geschke, Levin, Morris, Wick

Butler's message:

Automatic dereferencing: I don't see why your proposal requires automatic rightside dereferencing (as opposed to "dot" dereferencing); indeed, it seems more likely to require "referencing" (the procedure surely wants a pointer - that's the motivation for starting over again with x if it is a pointer originally). Am I missing something?

(1-3) Upon reflection, I decided I too preferred the more conservative position (that the record representing an object and all the operations be declared in the same interface). I guess this is easy for users to understand if they look upon the procedures in that interface as (potential) extra fields that have been moved out of the record. This position seems to minimize implementation problems and anomolies.

On adding new operations: this is probably a step in the wrong direction (building on kludges?), but I point out another

observation for what it's worth. Refer back to my example. If

D1: DEFINITIONS = ... T: TYPE = RECORD [val: D0.T']

it would be easy to check D1 for procedure P before stripping the record and going to D0. In this limited sense, T' and the operations in D1 "extend" T' and the operations in D0. Also, T' ← T is automatic, but not vice-versa.

(4) [also your reply to Peter's message]

It's true that the compiler currently doesn't have the information about which instance of an interface a type comes from. For reasons outlined in my previous message, I have already been looking into this problem. It appears that the bookkeeping changes are easy to arrange. If we accept this, however, it becomes hard to justify preserving any distinction between "included" and "imported" modules; the IMPORTS clause replaces the DIRECTORY, etc. We have discussed these changes before; they seem to make sense in the long run but cause some compatibility problems.

If the conservative view is adopted (so that single-component PRIVATE records do not cause anomolies), I believe that the sources of clients and implementers of a class have a good chance of surviving a change to a better object scheme, even for that class (if, in the case of implementers, a dynamically varying implementation is not needed). Obviously, the definitions won't have this property. I guess your scheme could continue to be supported in the glorious future; I am more worried about how it would interact with generic procedures, uniform references, etc., than with objects a la Peter Bishop.

Ed

Date: 27 Nov 1978 2:29 pm (Monday) From: Lampson Subject: Re: A modest proposal for objects in Mesa In-reply-to: Your message of 27 Nov 1978 11:53 am (Monday). To: Satterthwaite cc: Lampson, Deutsch, Horning, Mitchell, Geschke, Levin, Morris, Wick

The reason for wanting automatic dereferencing is this. Suppose I write x.P[y], and x turns out to be POINTER TO T, where P is in T's def module Tdefs, and is declared as PROC [xx: T, ...]. Then x.P[y] will be converted into Tdefs.P[x, y] and will then result in a type error. What is needed, in the absence of automatic dereferencing, is Tdefs.P[x↑, y].

Your comment on adding new operations seems to be going in the opposite direction from your preference for the more conservative position that all operations should be defined in the same place. Was it intended just as a curiosity, or an attempt to be fair to the other side, or what?

As you know, I am all for getting rid of the DIRECTORY / IMPORTS distinction.

A slight modification to my proposal would be to require writing OBJECT PROCEDURE (or something like that) for procedure declarations in an interface which are to be eligible for this wonderful transformation. This has the advantage of making the programmer's intentions clearer, and also of making it easier to automatically convert such source code into some other form in the glorious future.

Butler ---

Date: 2 MAR 1979 1652-PST
From: DEUTSCH
Subject: Documents on [Ivy]<CedarDocs>lang>
To: Cedar Language Features Design Committee:

I have converted to memo form both Jim Horning's message to the lab listing the unsafe areas of Mesa (jjh-2-27-79.memo) and the exchange of messages concerning Butler's "conservative" proposal for objects (bwl-objects.memo). I will also move Howard's and Peter Bishop's proposals onto this directory as soon as I can get hold of them. ---

Date: 4 MAR 1979 1558-PST
From: DEUTSCH
Subject: Minutes of 2/28 meeting
To: Cedar Language Features Design Committee:
cc: sturgis

Now on [Ivy]<CedarDocs>lang>2-28-79.minutes. I was able to make a little more sense out of the discussion retrospectively. ---

Date: 6 MAR 1979 1440-PST
From: DEUTSCH
Subject: Bishop's object proposal
To: Cedar Language Features Design Committee:
cc: sturgis

For any of you who haven't seen it, I've moved Peter Bishop's proposal for objects in Mesa to [Ivy]<CedarDocs>lang>bishop-objects.memo. ------

Date: 7 MAR 1979 1530-PST
From: DEUTSCH
Subject: Draft document on delayed binding
To: Cedar Language Features Design Committee:
cc: sturgis

The highly imperfect document that formed the basis of today's meeting may now be found on [Ivy]<CedarDocs>lang>delayedbinding1.memo. ------

Date: 23 Mar 1979 1:37 pm (Friday)
From: Horning
Subject: March 7 Minutes
To: Cedar-Lang↑

[Sorry for the delay--JJH.]

PRELIMINARY DISCUSSION

We agreed that the issues associated with object-oriented programming "belong" to the Delayed Binding Subcommittee; the Safe Language Subcommittee will need to be aware of any decisions that require apparent violations of the type system.

Satterthwaite: Free variables in types create problems for (among other things) the run-time type system.

Satterthwaite: As a design principle, we should insist that legality assertions be checkable EITHER by the compiler or by the run-time system (cf. Walsh paper on range checking in Pascal).

Bobrow: We may want a program quite separate from the compiler to infer, for example, ranges. Deutsch: Yes, BUT the certification of safety cannot be left to such programs--it is still a joint responsibility of the compiler and the run-time system, although hints provided by an inference program may simplify their job.

DELAYED BINDING (Deutsch)

Expressions and statements:

Must add INTERVAL values to the language.
Horning: As distinct from subrange types?
Deutsch: ... ?

Delayed binding of identifier interpretation:
ANY--defer binding type until assignment time.
FREE VARIABLE--defer binding access path until access time.
C/Mesa--bind-time specification of values.
Satterthwaite: Allowing procedures as L-values should buy you all three. Coercions create problems; most were banned by Mesa, but some have crept back in (e.g., widening).

Qualification of identifiers:
Selector bound at compile time.
We could postpone binding either
type of result, or
type of record to which applied.
Horning: Examples of use?
Deutsch: General principle: The programmer shouldn't have to think about whether a particular binding can be delayed.

Horning: Unease.

Applications as left sides:
Provide a syntactic mechanism to link corresponding Fetch and Store procedures.
Provide a run-time "procedural" interface to arrays.
Satterthwaite: I predict the re-discovery of Algol 60 "thunks."
Virtualize the notions "array," "variable access," "pointer," "iteration," "infix operator," so that the programmer may supply any implementation with the correct semantics.
The language must define the abstract operations (interface) that characterize each of the constructs, and be prepared to allow any suitable concrete implementation.
Virtual/Generalized pointer: operations are "Fetch through," "Store through," and possibly "Block move through."

Arithmetic and widening:
The language should support mathematical integers, possibly with more efficient implementations for expressions and variables whose range is known to be bounded.

Operators as abbreviations for applications.

Delayed polymorphism == object-oriented programming?

Mesa HAS a form of thunk--parameterless inner procedures.

Should we also allow call-by-name?
Deutsch: Call-by-name plus inline procedures gives "all" the benefits of macro expansion. [Mild skepticism]

Discussion to resume 10:30 Mon. Mar. 26.

Date: 13 Apr 1979 3:39 pm (Friday)
From: Mitchell
Subject: Global Variables in Mesa 5.0
To: MesaUsers↑
cc: SDSupport, Mitchell

This message is intended to encourage caution in the use of global variables in Mesa (i.e., in the use of interface variables in Mesa 5.0).

The indiscriminant use of global variables has a number of drawbacks, among which are the following:

(1) If a global variable, g, is available to be written by any module having access to the interface of which it is a part, and if the proper functioning of the module(s) implementing that interface depends on the variable, then the implementing module's integrity can be compromised by some other module changing g. If, instead, the interface contained two procedures Readg and Setg (possibly declared INLINE for efficiency), then the implementation of Setg can, in most cases, check that g is being assigned a "reasonable" value.

(2) If g is read before the module in which it lives has been started, the value is basically random because no start trap is caused, unlike calling a procedure in an unstarted module. For the same reason, a value assigned to g before its containing module has been started may be overwritten when the module is started. This particular form of bug has already occurred in at least one large subsystem.

(3) When one uses a procedural interface to access/change a global variable, one can set breakpoints on the procedures involved to catch all instances of setting (or reading) the value. The same is not true of exported variables. Thus, they are not only more likely to require debugging to discover erroneous assignments, but make it harder to debug as well!

Finally, I hypothesize that it is unlikely in a good interface design for it to be desirable to access/set a variable of the implementation of that interface - it certainly violates the concept of "information hiding". Consequently, I would expect the most common arguments for using global variables to be performance-related, and INLINE procedures answer those arguments perfectly adequately (indeed, it was inline procedures that forced the introduction of global variables in Mesa 5.0). Using inline procedures at least avoids problem (1), which I consider to be the major item.

Unfortunately, using inline procedures does not solve problem (2) because the expanded code of an inline procedure will not, by itself, cause a start trap, so it may access an uninitialized variable (i.e., the problem reappears one level removed). Until a solution for (2) appears, beware!

Jim Mitchell

Date: 14 APR 1979 2335-PST
From: DEUTSCH
Subject: Interface variables in Mesa 5.0
To: mitchell
cc: bobrow, horning, morris, satterthwaite, deutsch

Observation: if (as Danny's and my proposal for Objects suggests) we go back to the original Mesa notion of interfaces as records, then there is no reason why we can't have "fields" of the interface which are syntactically variables but actually procedures. We want that for other records anyway, I think.

The start trap problem, and some related ones, can be solved at the expense of one more level of indirection in the exporter: the indirect pointer can be initialized to NIL when the exporter's frame is allocated, which presumably will cause a trap, and replaced by the actual address of the variable just before the start code is run. ------

Date: 15 APR 1979 1927-PST
From: DEUTSCH
Subject: Howard's comments on our Objects design
To: horning, morris, satterthwaite

Date: 15 Apr 1979 5:34 pm (Sunday) From: Sturgis Subject: Objects To: Deutsch, Bobrow cc: Mitchell, Sturgis

Here are some initial reactions to your memo: BD-objects.memo

at first reading it seems to solve a lot of the original problems with modules as object instances. (I still intend to put together a list of those original problems.)

2 issues bother me: the limitation to one abstract interface definition per def file, and the limitation to export exactly one

interface from a module

one definition per def file:

suppose you want to represent a graph. the graph will consist of nodes and arcs. Nodes and Arcs are to be abstract types. an arc needs to suply a procedure similar to "GiveMeYourSourceNode: PROCEDURE RETURNS[NODE]" and similarly a node needs to supply procedures similar to "GiveMeYourFirstArc: PROCEDURE RETURNS[ARC]" . These would be impossible to define, since it would require a loop in the defs files.

one exported interface per module

We find the following situation occurs (at present within the context of normal mesa interfaces) There will be a more or less generally used interface to some service, and a second interface which is fairly implementation dependent (used for control). for example, in Juniper we have a public interface to the disk code. We also have a simulation of the disk code, and it exports two interfaces: the public interface, and a control interface which is used to specify how much disk to simulate and in what way the simulation should proceed. This second interface would most definitely not want to be included in the public interface, since it is not even used in the normal system.

Date: 15 APR 1979 2144-PST
From: DEUTSCH
Subject: Exporting more than one interface
To: Sturgis
cc: mitchell, bobrow, horning, morris, satterthwaite, deutsch

The example you gave could be handled by the subclass mechanism we proposed -- the simulation would be a subclass of the public interface. If SimulatedDisk is a subclass of Disk, then it is still possible to make a run time check whether a given object being treated as a Disk is actually a SimulatedDisk (basically, by checking the code segment pointer).

The other problem, of wanting multiple definitions to implementing several interacting objects, is a harder one to whih I don't have any instant solution. ------

Date: 16 Apr 1979 10:15 am (Monday)
From: Mitchell
Subject: Re: Interface variables in Mesa 5.0
In-reply-to: Your message of 14 APR 1979 2335-PST.
To: DEUTSCH
cc: mitchell, bobrow, horning, morris, satterthwaite

Peter,

I think that there is a flaw in your argument, beginning, ..." the indirect pointer can be initialized to NIL when the exporter's frame is allocates, which PRESUMABLY will cause a trap, ..." As far as I know, the only way to "trap" on NIL is to use a compiler option which produces extra code to check for NIL whenever a pointer is used. But I like the idea.

Jim

Date: 30 Apr 1979 4:05 pm (Monday)
From: Horning
To: Cedar Language Features Design Committee: Bobrow, Deutsch, Horning,
Morris, Satterthwaite;
cc: CSL-NOTEBOOK
Subject: Safe Language/Processes

Attributes: informal, draft, technical, Cedar, Dorado, Lisp, Mesa, Programming research References: 79CSL-XXX Abstract: "No tricks are safe in the presence of processes." Jim Morris

The Safe Language Subcommittee (Horning, Morris, Satterthwaite) met this afternoon, and spent virtually all our time contemplating with horror the degree to which the presence of processes invalidate all our nice comfortable assumptions about storage management, reference counting, garbage collection, and language safety in general.

We reached no conclusions other than that this problem is an order of magnitude harder than we had previously realized. We feel the need of a tutorial on the Bobrow-Deutsch incremental garbage collector (or whatever B-D feel is the current state of the art and default candidate for use in Cedar), with particular attention to the actions that will have to be made atomic, and would like to put this on the agenda for next Wednesday's meeting.

We are currently considering the conjecture that unMonitored sharing of automatically deallocated storage by separate processes will have to be banned from the safe language. Aside from feeling that this is a rather stronger restriction than people are likely to be happy with, we aren't sure that there is any static way to enforce such a restriction without some rather drastic restructuring of Mesa.

Jim H.

Date: 1 May 1979 4:17 pm (Tuesday)
From: Satterthwaite
Subject: Minutes
To: Cedar-Lang↑

On [Ivy]<CedarDocs>lang>4-25-79.minutes

Date: 1 May 1979 2:34 pm (Tuesday)
From: Birrell
Subject: Binding in Cedar
To: CedarFiling↑,CedarComm↑

A document describing my unified view of C-mesa, runtime binding, and remote binding is in [Ivy]<CedarDocs>Comm>Binding.bravo

Please make comments and suggestions. At some stage this document (or some such) should be given to the Language design group.

Andrew

Date: 30 Apr 1979 4:52 pm (Monday)
From: Satterthwaite
Subject: Bobrow-Deutsch GC Reference
To: Horning, Morris

CACM (Sept. 1976), 522-526

Date: 3 May 1979 3:38 pm (Thursday)
From: Horning
To: Cedar Language Features Design Committee: Bobrow, Deutsch, Horning, Morris, Satterthwaite;
cc: Owicki, CSL-NOTEBOOK
Subject: 5-2-79 Minutes: Garbage Collection and Processes

Attributes: technical, Cedar, Dorado, Lisp, Mesa, Programming research References: a79CSL-XXXa Abstract: This meeting of the Cedar Language Features Design Committee was devoted to the interaction of processes and automatic storage deallocation. The "safe language" subcommittee (Horning, Morris, Satterthwaite) claimed that this interaction introduced substantially more complexity than they had previously realized; the "delayed binding" subcommittee (Bobrow, Deutsch) claimed that the problems were understood and the overhead tolerable. These two claims are consistent.

INTRODUCTION:

We started with a discussion of the Bobrow-Deutsch incremental garbage collection algorithm (CACM Sept. '76) as modified by recent improvements. Basically, this is a variation on a reference-counting scheme, which aims at cheaply (and safely) collecting "almost all" the safely collectable storage. It is intended to be backstopped by a conventional trace-and-mark and/or copying garbage collector that works (on explicit user request) only while the system is quiescent (i.e., with no concurrent processes modifying list structure).

The three main modifications to standard reference-counting are

1) Reference counts are stored in a separate hash table, rather than with the objects themselves. Since a count of one is dramatically more frequent than all other counts combined, objects with a count of one are not stored in the table--i.e., a

count of one is indicated by ABSENCE from the table.

2) Rather than hashing each object and updating its count every time a reference changes, transactions are instead ENQUEUED for later processing by the collector. This minimizes the delay and locking required for user programs, and allows various optimizations by the collector.

3) Pointers from stack frames are not included in the explicit reference count; rather, the collector must not free any object unless its reference count is zero AND it is not pointed to from the stack.

There are various other optimizations. E.g., atoms are simply made permanent, and not reference-counted at all.

"Hashlinks" will have to be institutionalized, and treated as equivalent to explicit pointers.

The collection procedure is conservative. Treating any superset of the pointers contained in the stack as pointers will only delay the reclamation of some storage, and cause no other damage. This fact can be used to advantage, particularly in dealing with temporary results saved in Process State Blocks without knowing their types. E.g., treat all doublewords as pointers.

Certain invariants--not all of them yet identified--will have to be preserved by the entire system to enable safe automatic deallocation. In SAFE regions, their maintenance will be ensured by a combination of compiler checks and compiler-generated run-time checks; i.e., safety is a compiler responsibility. In UNSAFE regions, their maintenance will be a programmer responsibility; again, some combination of reasoning about the program and run-time checking may be used. The important point--which has not previously been stressed--is that UNSAFE regions do not allow violation of the safety invariants; they merely allow more flexibility in the methods of maintaining them.

RESTRICTIONS ON THE MUTATOR (garbage generator) REQUIRED BY B.-D. (partial list):

-It must be possible for the collector to find (consistent versions of) all frames and all Process State Blocks. In addition to global frames and frames locatable by following links from PSBs, this includes local frames accessible via ports or pointer variables; we think that such frames should simply be "registered" with the collector.

-The collector must be able to identify (a superset of) all pointers (to automatically deallocatable storage) within each such frame or PSB.

-Enqueueing a transaction must be an atomic action (e.g., either a single instruction or protected by monitor locks).

-[Point needing resolution] It is certainly sufficient to make the activity associated with storing a pointer value into a (non-stack) pointer variable-- namely, enqueueing an increment for one object and a decrement for the other and changing the variable's value--into a single atomic action. It is also clear that if the actions are ordered properly the activity can be decomposed into at least two atomic actions (<enqueue increment; save old value; store new value>, <enqueue decrement>). Further subdivision may be safe; it is not clear that the finer grain would be worth the additional overhead.

-The frame allocator/deallocator (at least) will need to synchronize with the collector to ensure that frames remain in existence while being scanned.

OTHER CONSIDERATIONS:

-Logically, the collector needs access to a consistent snapshot of
1) the hashtable
2) the transaction queue
3) all active frames and PSBs. 1) is easy, since only the collector accesses (let alone modifies) the hashtable. 2) is also straightforward, since only the collector removes transactions. Thus, it need only remember the position of the head of the queue at the time of the snapshot. 3) is the problem, since a scan of all active frames and PSBs is potentially time-consuming, and can therefore not lock out all other processes. However, when other processes run, they can make essentially arbitrary changes, completely invalidating the collector's information. The present attitude is to treat preemption of the collector as a rare, but not unexpected, event. The collector must have some way of detecting that a frame has changed since it was scanned, and be willing to rescan if that happens. The situation is quite analogous to the "dirty bit" for pages (and we briefly discussed how to piggyback on that bit by locking down all frame- and PSB-containing pages for the duration of the scan); no completely adequate proposal surfaced.

Jim H.

Date: 3 May 1979 5:13 pm (Thursday)
From: Horning
Subject: 4-18-79 Minutes

To: Cedar-Lang↑

Butler's metaphor for uniform treatment of "objects" in Mesa was discussed again.

Ed raised Peter Bishop's objections to the Bobrow-Deutsch object proposal.

Peter (D.) indicated that it seemed appropriate to rework the B.-D. object proposal; perhaps that proposal can be generalized along the lines suggested by Butler's metaphor to deal with some of Peter(B.)'s objections.

We discussed the issue of allowing a single definitions module to define more than one interface; apparently all that is required is some mechanism for indicating what is to be public and private in each of the bed's produced.

Jim H.

Date: 8 May 1979 4:29 pm (Tuesday)
From: Lampson
Subject: Making things undoable
To: Cedar-lang↑, Cedar-user↑

In its last meeting, the user facilities group spent some time discussing the implementation of undoable actions. One of the conclusions was that it would be very nice to get some help from the compiler in making a version of a procedure in which certain state-changing actions are undoable. The proper definition of "certain" is not clear. At one extreme, one might like to call a recording procedure for every assignment (this corresponds approximately to Lisp's TESTMODE=T), though the utility of this is somewhat debatable. More likely, one would want to log assignments to global variables, or through pointers, or to specified arrays or pointer bases. I am offering this as grist for the language designers' mill.

Date: 9 May 1979 9:25 am (Wednesday)
From: Horning
Subject: Re: Making things undoable
In-reply-to: Your message of 8 May 1979 4:29 pm (Tuesday)
To: Lampson
cc: Cedar-lang↑, Cedar-user↑

Butler,

Would you believe recovery blocks, with microcode assist for the Recursive Cache?

Jim H.

Date: 9 May 1979 9:48 am (Wednesday)
From: Satterthwaite
Subject: Re: Making things undoable
In-reply-to: Your message of 8 May 1979 4:29 pm (Tuesday).
To: Lampson
cc: Cedar-lang↑

I second Jim Horning's suggestion that recovery blocks should be considered. Going back even farther, a paper by Bob Floyd seemed very interesting when it appeared and still seems worth some attention (Nondeterministic algorithms, JACM 14 (October 1967), 636-644).

Date: 8 May 1979 11:20 pm (Tuesday)
From: Morris
Subject: Type memo available
To: Cedar-lang↑

On cedardocs, jhm-typeparameters.memo

Date: 10 May 1979 12:16 pm (Thursday)
From: Horning
To: CSL-NOTEBOOK
cc: Cedar-Lang↑, Owicki
Subject: Two more wrinkles on the Bobrow-Deutsch incremental garbage collector

Attributes: informal, draft, technical, Cedar, Lisp, Mesa, Programming research References: a79CSL-???, ???'a, CACM Sept. 1976

Abstract: In attempting to formulate the invariants to be preserved by processes using storage to be managed by the B.-D. incremental garbage collector, we identified two modifications that may simplify implementation in a multi-process environment.

The B.-D. incremental garbage collector scheme is based on a method known to be safe (reference counting) modified in a number of ways to improve performance without sacrificing safety. In the presence of multiple processes, it is necessary to examine both the basic reference-counting scheme and each of the modifications to determine the invariants that must be preserved to ensure safety, and the amount of locking/synchronization required to ensure their preservation. Sue Owicki has been looking at this issue for Cedar. While discussing her preliminary formulation, we discovered two further wrinkles that look attractive for the multi-process case.

1) The Transaction Queue that is used for communication between the Mutator processes and the collector process can actually be maintained as a set of queues, one per process. This completely removes the need for synchronization among Mutators (avoiding opportunities for deadlock or performance degradation), and will probably simplify the invariant that each Mutator must satisfy for the Collector's correctness to be assured. We had previously believed that the updating associated with storing into a pointer would have to be a small number of atomic actions (probably no more than two), probably created by explicit monitor locks; it now seems likely that this requirement can be relaxed.

2) The Collector must effectively gain a "consistent snapshot" of all the stack frames and the Transaction Queue. Since scanning all stack frames is likely to be time-consuming, we are unwilling to make it atomic, even though on a uniprocessor we expect interruption of the Collector to be relatively uncommon. We had previously considered marking each stack frame (or each page containing stack frames) with a bit analogous to a "dirty bit," which would be set by the Collector when the frame was scanned, and reset when an assignment was made into the page. Under the new scheme, we want to ensure that

a) each Mutator's Transaction Queue is marked at the point corresponding to the time when the stack frames were scanned, and

b) no Mutator has run between the time its stack frames were scanned and the time when the Collector finished scanning.

To ensure this, we associate a bit with each (Mutator) process, ScanCurrent, which is set by the Collector, and reset by the process scheduler whenever the process is run. After scanning the frames for all Mutators, the Collector enters a loop:

```
disable interrupts,
check ScanCurrent for each Mutator,   --this should be fast
reenable interrupts,
IF some Mutators had ScanCurrent FALSE
THEN rescan the stack frames for those Mutators and REPEAT
ELSE EXIT   --consistent state snapshot obtained
```

This loop should converge quickly if preemption of the Collector by a Mutator is indeed an uncommon event. It is safe for the Collector to scan all other processes, so it may not be worth distinguishing at the system level between processes that may change pointers to automatically deallocable storage (Mutators) and those that definitely will not.

Jim H.

P. S. Afterthought and query to Mesa experts: the above assumes that a process can only change "its own" stack frames. If this is not true, then it will be necessary to rescan all stack frames after determining that a consistent snapshot has not been obtained.

Date: 10 May 1979 4:46 pm (Thursday)
From: Morris
Subject: Minutes 5/10/79
To: Cedar-langt, Owicki

The meeting was spent discussing my proposal for type parameters.

1. The limitiations on the mechanism were not clear. Specifically, the fact that no type expression is ever actually substituted for a type parameter was somewhat of a surprise. In general, it became clear to me that the idea has to be

explained more fully.

2. Peter proposed the extension to allow parameterized interfaces, solving (among other things) the difficulty mentioned at the end of the memo about free type parameters in results lists.

3. Danny asked several clarifying questions about the advantages of type variables over objects.

4. Similarities between this proposal and Alphard, CLU, Russel, and some ideas from Graham Williams were noted.

5. Jim made some cautionary observations about how type parameters had caused unexpected problems in Euclid.

6. Sue (quite rightly) raised the question of how important the whole business was, based on experience with the DOD yellow langauge.

7. Implementation problems wer breifly discussed.

8. The meeting ended with no clear conclusion, and my resolve to incorporate the discussion into an improved memo (forthcoming).

Date: 10 May 1979 5:03 pm (Thursday)
From: Horning
Subject: Re: Minutes 5/10/79
In-reply-to: Your message of 10 May 1979 4:46 pm (Thursday)
To: Morris
cc: Cedar-lang↑, Owicki

Jim,

The entire discussion was haunted by a tantalizing sense that somewhere out in the object--abstract type--type variable--parameterized type--polymorphism swamp there floats (just beyond our reach) a glimmering, beautiful unification.

Although this is the stuff that legends are made of, we should also consider the possibility that it is just spontaneous combustion of methane.

Jim H.

Date: 11 May 1979 9:23 am (Friday)
From: Morris
Subject: Re: Minutes 5/10/79
In-reply-to: Your message of 10 May 1979 5:03 pm (Thursday)
To: Horning
cc: Cedar-lang↑, Owicki

I think there is mostly quicksand out there. (But, like many others, I think I know a safe path!)

Jim

Date: 14 May 1979 1:51 pm (Monday)
From: Horning
Subject: Smalltalk and types
To: Cedar-Lang↑

This two-page memo is interesting because it points out that lack of a type system in Smalltalk has significant, identifiable, real costs.

---------- Date: 14 MAY 1979 1034-PDT From: INGALLS Subject: Memo about inferring Type To: Smalltalk Interest Group:

I just wrote a memo about inferring type in Smalltalk. It summarizes a discussion with LRG on May 10. If you're interested, but didn't get hardcopy in your mailbox, look at [MAXC]INGALLS>Type.Bravo. ------- ----------

Date: 16 May 1979 11:16 am (Wednesday)
From: Lampson
Subject: Objects
To: Cedar-Lang↑

I have written yet another memo on this subject. You can find it on [Ivy]<Lampson>Objects.memo.

Date: 16 May 1979 6:17 pm (Wednesday)
From: Horning
Subject: Re: Objects
In-reply-to: Your message of 16 May 1979 11:16 am (Wednesday)
To: Lampson
cc: Cedar-Lang↑

Butler,

I'm (at least temporarily) convinced that this makes sense, is safe, buys a lot, etc. What has Howard's reaction been?

In reading your exposition of "The basic idea," I was struck by the following question:
"Suppose a procedure has more than one parameter whose type is
abstract? Does the scheme go through?" and wasn't able to answer it to my own satisfaction. Can you?

Your aside on Polymorphic procedures triggered the following ramblings:

In the "object" case what is OK is assigning something that is known to be of any of the concrete types to a variable of the abstract type. It would NOT be acceptable to assign it to a variable of another concrete type that was also an implementation of the same abstract type, nor could we safely assign a variable of the abstract type to a variable of one of the concrete types without performing a discrimination of some kind.

In the "polymorphic procedure" case, what you want to prevent is assigning something of one concrete type (Integer) to a variable of another concrete type (Real); there is nothing wrong with assigning either one of them to a variable of the abstract type (Sortable). The issue here appears to be that the array to be sorted is both abstract (in the procedure body) and concrete (outside), and we want a simple check on the procedure body that ensures that concrete types will be preserved.

Perhaps the key difference here is the "sharing" (forgive the term) of the array variable by two type declarations (outside and inside) with different levels of specificity. I am reminded of the difference between ANY and PARAMETER for variant records (which are your source of inspiration) in Euclid. The array parameter has abstract type Sortable, and implicitly gets the concrete type tag PARAMETER (unchangable), whereas local variables of the same abstract type implicitly get the concrete type tag ANY (changable).

Is this a clarification, or just a red herring?

Jim H.

Date: 18 May 1979 12:13 pm (Friday)
From: Horning
To: Cedar Language Features Design Committee: Bobrow, Deutsch, Horning,
Morris, Wood, Satterthwaite:, Sturgis
cc: CSL-NOTEBOOK
Subject: More on "Yet another view of objects"

Attributes: informal, technical, Cedar, Mesa, objects References: 79CSL-XXX (Lampson memo of May 14, 1979)

Abstract: Abbreviating puns in Lampson's memo made it less than crystal clear exactly how general his proposal is. Danny Bobrow and I now feel that his proposal essentially bridges the gap between implementation and the "dictionary oriented" approach that Bobrow and Deutsch have advocated.

INTRODUCTION

Danny Bobrow and I spent about an hour this morning trying to determine whether Butler's proposal for another view of objects covered all the cases of interest to Danny, and after much thrashing around came to the conclusion that it did, although the reasons were not obvious in Butler's memo. It seems worth recording our conclusions and suggested clarification.

For purposes of discussion, suppose that our abstract type is Set, with concrete implementations CS1, CS2, ... Consider the operation Union. We can imagine several styles of Union operation:

-Fully generic: Union[s1, s2: Set] RETURNS [s3: Set]

-Semi-generic: Union1[s1: CS1, s2: Set] RETURNS [s3: CS1],
Union2[s1: CS2, s2: Set] RETURNS [s3: CS2], etc.

-Concrete: Union1[s1, s2: CS1] RETURNS [s3: CS1],
Union2[s1, s2: CS2] RETURNS [s3: CS2], etc.

Danny's question was: Does Butler's proposal allow us to choose among these styles, and if so, how do we specify the choice?

>>>>>>THE PROBLEM

We lost quite a lot of time, we feel, because Butler's proposal contains abbreviated notation that obscured the answer to that question, which we now feel is "Yes."

We believe that the example starting on the last line of page 2 of Butler's memo should properly have been written

```
O: TYPE = ABSTRACTION
    POINTER TO POINTER TO RECORD [
        A[p: O, ... ] RETURNS [r: O],
        B[p: O, ... ]  ]
    WITH [
        SELF.C[ ... ] CALLED BY SELF↑↑.A[SELF, ... ],
        SELF.D[ ... ] CALLED BY SELF↑↑.B[SELF, ... ]  ]

I1: TYPE = IMPLEMENTATION OF O
    POINTER TO RECORD [
        procs: POINTER TO RECORD [
            A[s: I1, ... ] RETURNS [t: I1]
            B[s: I1, .... ] ]
        data: whatever ]
    WITH [
        SELF.C[ ... ] CALLED BY SELF↑.procs↑.A[SELF, ... ],
        SELF.D[ ... ] CALLED BY SELF↑.procs↑.B[SELF, ... ] ]
```

>>>>>>THE SET EXAMPLE - FULLY GENERIC

Union would not appear in the type declaration at all--it would merely be declared to take two Set parameters and return a Set result.

>>>>>>THE SET EXAMPLE - SEMI-GENERIC

```
Set: TYPE = ABSTRACTION
    POINTER TO POINTER TO RECORD [
    ...
        Union[s1, s2: Set] RETURNS [s3: Set]
    ... ]
    WITH [
    ...
        SELF.Union[s5: Set] CALLED BY SELF↑↑.Union[SELF, s5]
    ... ]

CS1: TYPE = IMPLEMENTATION OF Set
    POINTER TO RECORD [
        procs: POINTER TO RECORD [
        ...
            Union1[c1: CS1, c2: Set] RETURNS [c3: CS1]
            --Within this concrete procedure, c2 can only be
            --manipulated as a Set (i.e., abstractly).
        ... ] ... ]
    WITH [
        ...
```

```
        SELF.Union[c5] CALLED BY SELF↑.procs↑.Union1[SELF, c5]
        ... ]

>>>>>>THE SET EXAMPLE - FULLY CONCRETE

Set: TYPE = ABSTRACTION
    POINTER TO POINTER TO RECORD [
    ...
        Union[s1, s2: Set] RETURNS [s3: Set]
    ... ]
    WITH [
    ...
        SELF.Union[s5: Set] CALLED BY IF SELF↑ = s5↑
                        THEN SELF↑↑.Union[SELF, s5]
                        ELSE GenericUnion[SELF, s5]
    ... ]

CS1: TYPE = IMPLEMENTATION OF Set
    POINTER TO RECORD [
        procs: POINTER TO RECORD [
        ...
            Union1[c1, c2: CS1] RETURNS [c3: CS1]
            --Within this concrete procedure, c2 can be
            --manipulated as a CS1 (i.e., concretely).
        ... ] ... ]
    WITH [
        ...
        SELF.Union[c5] CALLED BY IF SELF↑ = c5↑
                    THEN SELF↑.procs↑.Union1[SELF, c5]
                    ELSE GenericUnion[SELF, c5]
        ... ]
```

Jim H.

Date: 21 May 1979 3:31 pm (Monday)
From: Morris
Subject: Yet more Comments On Objects
To: Cedar-lang↑, Lampson, Sturgis, Wood
cc: CSL-notebook

I would like to see more examples of how one can exploit the generality of Butler's proposal. At the moment it sems to be generalizing from a single example, Humus objects. It doesn't seem to be much of an advance over the current (verbose) way Humus objects are implemented except that the compler checks that the two records of procedure values are compatible in some sense. I am also a little disturbed by the idea of letting the notion of type compatibility depend upon coincidences in what code the compiler generates; even if the compiler never changes it is a little hard to explain to someone who doesn't think about the compiled code in detail.

On polymorphic procedures .

The polymorphic sort procedure is not a big problem.

First of all, if you are willing to pass both a Comparison routine and a Swap routine, you can bundle them both into an object called SortObject.

```
SortObject: TYPE = RECORD[Swap: PROCEDURE[i, j: CARDINAL],
                                                Less: PROCEDURE[i,j: CARDINAL]]
                                        RETURNS[BOOLEAN];

PROCEDURE Sort[so: SortObject, N; CARDINAL];
```

The Sort routine cannot even get a value in its hand to do the things Butler was worried about. Of course it is restricted in its methods; it can't

The real problem comes when one wants to supply just a Comparsion Routine and then have the Sort routine handle the values in a gingerly way. To do this one must introduce (perhaps implicitly) a type parameter and then use the scope of this type parameter to prevent polymorphic values from escaping from the limited context. The example in my type parameters memo does this:

Sort: PROCEDURE[t: TYPE, A: ARRAY[0..1) OF t, Less: PROCEDURE[t,t] RETURNS [BOOLEAN], N: CARDINAL];

Why is this safe in the presence of processes and co-routines? First notice that anything (variable, array, procedure, signal, port,...) declared outside of Sort cannot even mention the type t. Second, two instances of Sort could communicate between each other only by reference to some such external thing or by recursion. No value of type t can move through the external things, so we need only worry about recursion. But that is not a serious problem either; the two instances can only communicate through Sort's parameter list which introduces the formal parameter t, preventing the callee from talking about the old one.

The sort procedure is something of a special case, however, as its polymorphism is so transient. There seem to be cases where one wants to have two concrete implemetations running around in the same context and still have the type checker keep them sorted out. In the Set example that Danny and Jim have tussled with the client of the two implementations may not want to have them mixed up.

To make things distinct I would like to have the equivalent of a parameter list to be able to bind things to so that I could say

```
LET [Set1: TYPE, ..., Insert: PROCEDURE[CARDINAL, Set1] RETURNS [Set1],
                              Union1: PROCEDURE[Set1, Set1] RETURNS [Set1], ...]
    = SetImplementer1[];

LET [Set2: TYPE, ..., Insert1: PROCEDURE[CARDINAL, Set2] RETURNS [Set2],
                              Union2: PROCEDURE[Set2, Set2] RETURNS [Set2], ...]
    = SetImplementer2[];
```

These two declarations introduce two distint types (even if SetImplementer1 = SetImplementer2 !)

I am not sure how to combine these into a single type, but think it is a job for the dynamic type system since in Jim and Danny's soultion to this problem they do some explict testing of run-time types.

The need to keep things distinct is more compelling in the polymorphic List example in my memo. There it essential to keep lists of one type distinct from lists of the other.

Jim

Date: 23 May 1979 9:57 am (Wednesday)
From: Morris
Subject: New memo on types
To: cedar-lang↑, owicki, lampson

On cedardocs>lang>jhm-polytypes.memo is an idea for reconciling runtime and compiletime checking. Everything beyond Example1 is in rather confused state, so don't try to understand it.

Jim

Date: 25 May 1979 11:25 am (Friday)
From: Horning
Subject: Simplifying the Concurrent Garbage Collection Problem
To: Cedar-Lang↑
cc: Owicki, Popek

This morning I mentioned to Jerry Popek that the more we looked at the problem of Cedar garbage collection, the more tricky and special-casey it got. Jerry's response was "If you are looking for a research problem, that's fine, but if you just want to get the job done, you're solving too hard a problem."

In particular, the problem that Jerry proposed to brute-force is the interruption of the "snapshot" phase of the collector by a higher-priority process. His reasoning is as follows:

There are very few processes whose response times must be such that they can't afford to wait the fraction of a second that the snapshot will take.

These processes are typically device drivers, small, and in Unsafe Mesa, anyhow.

We can avoid the need to snapshot them by any of several means, e.g., prohibit such processes from having any direct pointers to collectable storage, requiring that they only point to collectable storage that is also KNOWN to be pointed to from a collectable process, or simply prohibiting them from CHANGING a pointer to collectable storage.

I agree that we should avoid as much research as possible here; the problem has enough tricky special cases without dealing with any we don't have to!

Jim H.

Date: 30 May 1979 11:20 am (Wednesday)
From: Horning
Subject: Butler on GC Simplification
To: Cedar-Lang↑
cc: Owicki, Popek, Lampson

I had a long talk with Butler just before the long weekend, in which I expressed my disappointment that getting to the bottom of the garbage collection problem for Cedar was proving harder and more time-consuming than anticipated. Butler strongly endorsed Jerry's suggestion that we attempt to evade most of the problems, rather than solving them (i.e., vigorously apply the "avoid unnecessary research" principle).

The following suggestions have been culled from my cryptic notes of that conversation:

-Butler agrees with Jerry that it is acceptable to simply disable interrupts during the "stack scan" phase, thereby greatly simplifying the job of getting a consistent snapshot.

-He also suggests that there is no reason not to microcode "assignment to variable containing pointer to reference-counted storage," thereby assuring atomicity.

-The combination of these two simplifications means that there is no longer any reason to keep a separate queue for each process.

-We can get along for quite a while without doing anything special for circular reference-counted structures. Danny's refinements can be added at leisure.

-Butler suggests that we adopt the suggestion of making the units of allocation and deallocation the same (i.e., pointers to collectible data must always be copies of the pointers handed out by the allocator, not pointers to substructure).

-To make this restriction tolerable, Butler suggests treating VAR parameters specially DURING THE STACK SCAN. This simply means that we continue to hold on to data with zero reference count if a VAR parameter points anywhere within it (presumably a relatively cheap extension to the check we have to do anyhow).

-We can distinguish within the language between pointers to collectable storage and pointers to "other" storage (e.g., stack variables and explicitly-freed heap storage), and place different restrictions on each. This may also provide some small efficiency in storage structure tracing (e.g., we can stop whenever we reach something that cannot point directly or indirectly to collectible storage). Many uses of conventional Mesa pointers will, of course, be limited to the unsafe subset.

Comments?

Jim H.

Date: 30 May 1979 11:40 am (Wednesday)
From: Horning
Subject: Getting it done (and documented) in June
To: Cedar-Lang↑
cc: Lampson

At the rate of progress in the last month (positive in terms of understanding and research, negative in terms of nailing down a design), it is going to be very difficult to meet our internal target of a complete draft design, ready for criticism outside the committee, by the end of June.

I propose that we restrict our ambitions for Mesa 5C1 rather radically, so that we arrive at something solid and well-understood, with the agreement that that will not be the end of the line. More specifically, I suggest

-Garbage collection: We adopt all the simplifications of my two recent messages; the result will still be a quite usable advance on what Mesa now offers, and will still present adequate implementation challenge.

-Safe language: We eliminate all the features identified as unsafe from the safe subset, but do NOT initially provide

many of the "safe substitutes." (Possible exceptions: VAR parameters, LOOPHOLEs not involving production of pointers [PUNs?], "Butler's Objects.") We use the Mesa 5C1 experience we get as guidance for consideration of further substitutes in 5C2.

-Delayed binding: We attempt to merge (a conservative subset of) the better features of Peter and Danny's latest proposal, Butler's objects proposal, and Andrew's Interface Record proposal, paying considerable attention to smooth meshing and comfortable syntax. We postpone any features whose interactions with other features or implementation seem difficult to Mesa 5C2.

-Type system: We use POINTER TO ANY and abstract interfaces as our only relaxations of strict static typing in Mesa 5C1. Specifically, we defer until 5C2 or later the fascinating research topics that Jim Morris has recently raised concerning the interactions of static checking, polymorphism, and types as variables or parameters.

I think that we could conceivably complete a design for Mesa 5C1 and get a reasonable document in a month, if we adopt a set of restrictions similar to those sketched above. I am open to suggestions about other restrictions that might have a higher payoff/cost ratio. I am (at least currently) very down on pursuing any research breakthroughs that appear within our grasp.

Jim H.

Date: 30 May 1979 1:08 pm (Wednesday)
From: Satterthwaite
Subject: Re: Getting it done (and documented) in June
In-reply-to: Your message of 30 May 1979 11:40 am (Wednesday)
To: Cedar-Lang↑
cc: Lampson

I agree with Jim Horning that the end of June is approaching rapidly. Here are some comments on his suggestions.

-Garbage collection: It seems to me that we still need a language extension to recognize and "bless" the allocation of non-frame variables. There has been a design for this kicking around for quite some time, but it has never seemed worth implementing for current applications without some notion of "generalized pointer". Are we agreeing to limit our goals to 32-bit (and MDS-relative 16-bit?) absolute pointers?

-Safe language: I am still not very convinced of the wonderfulness of VAR parameters. Parameter passing using a pure "value" semantics seems to generalize more nicely to remote procedures and coroutines (see Andrew's proposals for the former; a recent CSRG report from Toronto by Vanek illustrates some ugly interactions between VAR parameters and coroutines). If VAR parameters are to be provided, however, another alternative is to represent each VAR parameter by a pair (pointer to allocation unit, offset). This has the side benefit of allowing part-word variables as VAR parameters (by making the offset a byte pointer). I also vote for "persistent closures" in 5C1. I think the payoff will be very high when people learn to use them, even without type parameters.

-Delayed binding: The B-D objects proposal struck me as elegant, understandable, reasonably conservative, and more than adequately general for 5C1. It seems to me that there shouldn't be much impact on clients using such objects even if we achieve object nirvana in 5CX.

Date: 31 May 1979 3:36 pm (Thursday)
From: Horning
Subject: Pointing vs. Naming, etc.
To: Cedar-Lang↑
cc: Lampson, Owicki

This message notes a number of afterthoughts and clarifications to yesterday's discussion concerning, pointers, references, VAR parameters, safety, and the like.

ON SAFETY

INVULNERABILITY is the (obviously desirable) property of a SYSTEM that ensures that programming errors cannot reduce its world to a rubble of bits. To ensure it, certain INVARIANTS must be maintained throughout the system.

We have been using the term SAFE to denote program regions where the compiler is able to enforce the PRESERVATION of the invariants. If the invariants are initially true, and the entire system is safe, then it is also invulnerable. However, if it contains any unsafe regions, then all parts of the system are VULNERABLE to programming errors in the unsafe regions. SAFE thus merely implies that certain types of (local) checks are made automatically, UNSAFE, that the programmer is responsible for making them. The safe/unsafe boundary does not provide a firewall to limit vulnerability, only responsibility.

REFERENCES AND POINTERS

We decided that a good, pronouncable spelling of
POINTER TO COLLECTIBLE t would be
REFERENCE TO t Furthermore, this is quite consistent with common usage (reference counts, etc.). [Note that this implicitly ties the COLLECTIBLE property to the pointer/reference type, rather than to the type t.]

To first approximation, pointers and references must address disjoint storage (i.e., it makes little sense to account for some, but not all, the ways an object can be accessed). We also decided that it makes good sense to use the units of allocation as the units of deallocation. In particular, this means that the ONLY way in the language to get a new reference is by the NEW operation, and when a reference count goes to zero the entire item referenced can be freed. The most important restriction this introduces is that we are disallowing direct references to substructures. (However, see below.) Reference and pointer types are completely incompatible; in particular, they are NOT interassignable.

Pointer operations will be essentially the same as in present Mesa (but, we hope, used mainly in unsafe regions!). The @ operator is definitely unsafe. By disjointness, it cannot safely be applied to collectible items or their components (but, of course, it cannot be in applied present Mesa, at all!).

The collector must know about pointers (and the types they point to), and be prepared to follow them to find references, just as it must know about other things--such as the stack--that it cannot deallocate, but must logically trace.

INTERNAL REFERENCES

Conceptually, we can imagine implementing a reference to a component of a collectible structure as a [Reference, Offset] record (with some fiddling behind the scenes to keep the types straight). This preserves the property that we reference only directly allocated units, and don't create references except by NEW. Raw internal references are not proposed as a language extension, rather, they provide a convenient way of viewing the semantics of the concepts discussed in the next section.

Non-local variable references can be thought of as having this form; they are "internal" to a (global or local) frame. Thus, persistent closures can be easily unified with the following proposals.

NAMING VS. POINTING

[This section was triggered by further consideration of Danny's request not to create a special VAR mechanism JUST for parameters, and by an attempt to map yesterday's discussion into Euclid terminology and concepts; I finally realized that the mechanism was already in Mesa, EXCEPT for parameters.]

We frequently want to be able to NAME substructures just as though they were variables: Passing a substructure as an actual parameter to be named by the corresponding formal parameter is the canonical example. I believe, however, that a real need to have a variable POINTER or REFERENCE to a substructure is very rare. What appear to be instances of this need are actually uses of pointers/references as readily available substitutes for dynamically bound names.

Euclid has a uniform mechanism for binding a name to a variable reference, which is the same in a parameter list or a declaration list. I'm not in love with the syntax, but we worked out the semantics very carefully--and, I believe, correctly. Basically, in the terminology we have been using, it is possible to dynamically bind a name (identifier) to any (component of a) variable that can be written at the corresponding point in the program; the expression computing the variable can be of arbitrary complexity, and is evaluated once and then preserved for the lifetime of the scope.

Conceptually, each binding is implemented by creating an anonymous local "internal reference" constant that is initialized by the expression, and then supplying this constant, dereferenced one level, for each use of the identifier. Since the constant ITSELF cannot be named within the program, there is no way to either assign to it or copy it (i.e., no transactions of interest to the collector can be generated). THIS IS THE KEY PROPERTY THAT MAKES BINDINGS MORE TRACTABLE THAN REFERENCE VARIABLES, and hence of considerable interest to us. Furthermore, since all bindings appear on the stack, it is perfectly safe to allow the "reference" component to actually be a "pointer," i.e., to be the address of a stack or other non-collectible variable. This is the only case where I propose to tolerate a pointer/reference ambiguity. Note that the ambiguity cannot "spread" or "infect" reference variables, since bindings are not assignable.

[Mesa's OPEN can be viewed as a binding in which the name may be "empty." (In my opinion, omission of the name should be discouraged.) Note that this means that OPENs are of interest to the collector, since they increase reference counts. However, the Manual appears to (unnecessarily) restrict the use of OPEN to variables of type record.]

VAR formal parameters are conceptually just ordinary bindings, except that their initialization is supplied at the point of call, by the actual parameters, rather than at the point of declaration. The implicit creation of a pointer to a variable in a calling sequence replaces the principal safe use of @; this makes me comfortable with the restriction that totally bans @ from the safe language. [Note that it is the CALLER, not the procedure itself, that determines whether a reference is

being coerced to an internal reference, a reference to a substructure is being created (this includes the important case of a variable within a stack frame), or a pointer is being coerced to an internal reference.]

SUMMARY AND PROPOSAL

My current feeling is that we should:

-Generalize OPEN to arbitrary types, and eliminate the "empty" name feature (at least in the safe language).

-Find a pleasant syntax for VAR parameters that stresses their semantic equivalence to OPEN.

-Allow internal references to occur only in OPENs and VAR parameters.

-Allow OPENs and VAR parameters to bind names to dereferenced POINTER and REFERENCE values.

-Disallow @ in the safe language.

-Disallow freeing of heap storage in the safe language.

-Treat REFERENCE TO t and POINTER TO t as distinct types that cannot be interassigned.

-Provide NEW as the ONLY way to generate new reference values. In particular, provide no way to generate references to substructures (including stack and heap variables).

-Incorporate initialization of the allocated storage as an integral part of the NEW operation (as in Euclid).

COMMENTS WELCOMED!

Jim H.

CSL Notebook Entry

To	CSL	Date	May 14, 1979
From	B. W. Lampson	Location	PARC/CSL, Palo Alto
Subject	Yet another view of objects	File	[Ivy]<Lampson>objects.memo

XEROX

Archive ID: yyCSL-xxx

Attributes: informal, technical, Cedar, objects

References: None

Abstract: Objects with multiple implementations can be viewed as a sugaring of variant records and select statements. This view makes clearer the relation between such objects and the type system.

I offer here a new way of looking at objects with multiple implementations, which shows that they can be regarded as a clever and comfortable packaging of variant records and their associated discriminations. To begin, I will show how to provide fully general objects in Mesa 5, with absolutely no extensions and no loopholes. This scheme is not one which anyone would actually want to use, for three reasons:

it requires a lot of ugly and redundant writing;

it requires all the variants of the object to be explicitly declared in one place;

it generates a lot of unnecessary code.

Next I show how the third objection can be overcome with some cleverness ont he part of the compiler, but still without any language extension. Finally, I offer a (skeletal) language extension to overcome the first two objections.

The basic idea

Suppose that I have concrete types I1 and I2 which are intended to be instances of the abstract type O with procedures

A[O, ...] RETURNS [O] and B[O, ...].

The corresponding procedures in I1 and I2 are

A1[I1, . . .] RETURNS [I1], B1[I1, . . .],
A2[I2, . . .] RETURNS [I2], B2[I2, . . .].

Of course, the other parameters to these procedures must have the same types, and the results must have the same types, except possibly for occurrences of the concrete type. Here is how to define O and call its procedures

```
O: TYPE = RECORD [ SELECT * FROM
       i1 => I1,
       i2 => I2
       ENDCASE ]
```

When you want to write O.A, instead write

```
WITH dO: O SELECT FROM
        i1 => i1[A1[dO, . . .]];
        i2 => i2[A2[dO, . . .]];
        ENDCASE
```

If the abstract procedure returns something which is not of the abstract type, the constructors in the selection are not needed. If it returns something which has the abstract type embedded in a more complex structure, then a more complex constructor is needed.

Claim: this construction has exactly the desired effect, in terms of turning the two (or many) concrete types into a single abstract type. It has no loopholes and uses only standard Mesa. In fact, it would still work even if entirely different code sequences were required for calling A1 and A2, as long as the parameters required are contained in the parameters of A. For the rest of the scheme to work, however, it is essential that exactly the same code sequences be used for all the cases.

Saving code

An optimizing compiler can recognize that it is generating exactly the same code for each arm of the SELECT statement, and that all the alternatives have been enumerated in the selection; therefore the selection is not needed. As a consequence, exactly the same code will be generated as we would have gotten by loopholing the I*j* into O; not only will there only be one copy of the code to call the procedure, but the tag field of the variant record will never actually be examined at all. If the compiler can examine all the places in which O is used and see that they all have this form, it can omit the tag field entirely.

True abstract types

You are now supposed to be convinced that the proper way to introduce an abstract type is with a language extension which specifies the procedures and the code to be generated in calling them. Then any concrete type can be an instance of this abstract type if plugging its procedures into the (now imaginary) select statements of the form given above will result in a program which

is legal (i.e., passes the type-checker), and

generates exactly the same code that the abstract type specifies.

This must work for the same reason that the construction above works; a new type which comes along will be accepted only if the decisions which the compiler has already made (in compiling a lot of users of the abstract type) are "by accident" correct for this new implementation.

This is a simple test which can easily be made by the compiler when the concrete type is defined. If I1 is an instance of O in this sense, it means that the assignment

o ← i1 .

is legal.

With this extension, it is no longer necessary to define a single variant record for all the implementations (since in fact no variant record at all is required), and the compiler can be expected to take care of interpreting O.A correctly.

Thus, the declaration of O for a Humus-style way of accessing the procedures (see the appendix if you are not familiar with Humus) might look like this

```
O: TYPE = ABSTRACTION
```

```
POINTER TO POINTER TO RECORD [
        A[O, . . .] RETURNS [O],
        B[O, . . .] ]

WITH [
        A[O, . . .] RETURNS [O] CALLED BY O↑↑.A[O, . . .],
        B[O, . . .] CALLED BY O↑↑.B[O, . . .] ]
```

A concrete instance can be declared with

```
I1: TYPE = IMPLEMENTATION OF O

        POINTER TO RECORD [
                procs: POINTER TO RECORD [
                        A[I1, . . .] RETURNS [I1],
                        B[I1, . . .] ]
                data: whatever ]

        WITH [
                A CALLED BY I1↑.procs↑.A[I1, . . .],
                B CALLED BY I1↑.procs↑.B[I1, . . .] ]
```

Obviously this leaves a good deal to be desired in convenience, but I think those are straightforward syntactic issues. Note that the two declarations do *not* have exactly the same form; only the "accident" that o↑↑.A generates the same code as i1↑.procs↑.A makes it work. This is one of the things the compiler must check in accepting I1 as an instance of O.

Polymorphic procedures

There is a strong superficial resemblance between objects with multiple implementations and polymorphic procedures (e.g. a *Sort* routine which can sort arrays of any type T with *Compare* and *Swap* procedures). In both cases the type (O or T) is known only through the procedures which can be invoked on it. There is also, however, a fundamental difference (or so it seems to me this week). In the case of *Sort*, it is essential that different T's don't get mixed up. If there were two instances of *Sort* running simultaneously (in two processes, or with co-routines, or whatever), one with T=*Integer* and the other with T=*Real*, and somehow an *Integer* T got assigned to a *Real* T, this would be a disaster. The reason is that the various T's make no pretense of being interchangeable, except the the narrow purposes of the *Sort* routine. In the case of the objects, however, the whole point is that they *are* interchangeable; it is just fine to assign one O to another. This works because the discrimination is done only at the last minute, when one of the operations is called.

Appendix on Humus

Humus used the convention that an object is a pointer to a two word record. The first word of this record is a pointer to a record of procedures, and the second word is a pointer to a data record. The abstract type only cares about the procedure record, and considers the types of the procedures to be PROCEDURE [abstract, ...]; the concrete type cares about both records (since it folows the data pointer to access its data) and considers the types of the preedures to be PROCEDURE[concrete, ...]. A loophole is required to make the concrete type into an abstract one.

Inter-Office Memorandum

To CSL

Date May 31, 1979

From Cedar Database committee
(Cattell, Deutsch, Popek, Suzuki)

Location PARC/CSL, Palo Alto

Subject Cedar Database Concepts and Facilities

XEROX

File: [Ivy]<CedarDocs>database>ConceptsFacilities.memo

Archive ID: yyCSL-xxx

Attributes: technical, Cedar

Abstract: The Cedar Database project is intended to provide a convenient, high performance data management facility for use by all Cedar clients. The goal is an integrated, unified representation of data throughout a network, facilitating information exchange and avoiding repeated implementations of specialized data handling facilities. Here we present the basic concepts and capabilities projected for the Cedar Database facility.

Concepts and Facilities for Cedar Databases

I. Introduction

The Cedar Database project is intended to provide a convenient, high performance data management facility for use by all Cedar clients. The goal is an integrated, unified representation of data throughout a network, facilitating information exchange and avoiding repeated implementations of specialized data handling facilities. Once such a facility is in place, it should in many cases not be necessary for clients to deal directly with the network, as a sophisticated optimization and caching facility is planned. Instead, clients see a net-wide database, with some uniform naming scheme so that data items can be referenced directly; FTP or hand overlaying is not necessary. Further, the multiple layers of software inevitably needed to support these functions will be optimized to a considerable extent by compilation strategies.

A number of potential clients of a Cedar database facility were examined: specific characterizations of them can be found in the appendix. In general, there are a large number of applications, with fairly similar needs. They are mostly reasonably small in terms of the number of stored data items, and bi-modal in terms of performance needs: A.I. applications need very fast access, while program storage requirements are more modest. Physical integrity of the stored information is very important. Simple solutions to distributed data management problems will suffice. Protection against malicious uses is not important, and simple synchronization facilities based on Juniper are expected to be satisfactory. The Cedar database design has been guided by the results of this survey.

We envision three semantically different interfaces to the data base: the *low level* interface, which deals with the actual storage representation of objects; the *view level* interface, which still deals with individual data objects but provides data independence; and the *high level* interface, which deals with general queries. The low level interface is internal to the data base system: clients never see it and cannot manipulate its contents directly.

The Cedar facilities will not include a "high level" query language, such as the non-procedural query-by-example, or other interactive database interfaces. Only a "low level" storage mechanism, augmented by a means of data abstraction called views, will be provided. This decision was made both to limit the (still ambitious) level of aspiration and to avoid what were expected to be

diverging ideas of what the high level should look like.

The view facility is the main source of *data independence*, a central concept in data management. Briefly, the goal is to permit clients to write their programs using some logical model of the stored data, while permitting significant reorganization of the actual stored data without necessitating *any* resulting changes to those programs. The importance of data independence in avoiding terribly painful and often impractical revision of software is now well recognized in the data base community.

This document outlines the functions that are intended for the Cedar Database, together with a general implementation approach. At the end, an example of how a client might use the facilities is outlined.

In a number of respects, the Cedar database facility is patterned after the lower levels of System R, a relational database developed at IBM San Jose. System R is the only successful system we know of which 1) has multiple defined interfaces, at different levels; 2) is able to support different logical models; and 3) has transaction and locking mechanisms that fit our needs. In our distributed, decentralized computing environment, of course, IBM's solution of centrally maintained locking and allocation facilities is not appropriate; also, since we expect data base accesses to be an integral part of many of our applications (and of our programming tools), we must plan for a very high access rate. We know of no successful implementation of such data base facilities. Therefore, we have studied the facilities of System R to decide how to adapt them, while also dealing with the usual distributed-system problems of redundant storage and communication overhead.

In our case, there is also much more integration of views and the programming language by which the database is to be accessed; database objects are accessible from Mesa directly rather than requiring the usual "pasting" job. We hope to support the relational, hierarchical and network models directly. Compilation will also be used more extensively, as our performance goals are demanding.

II. Low Level (Storage Level) System Functionality and Architecture

The low level consists of procedures which can efficiently (that is, using the storage representation directly) manipulate the objects listed below.

Primitive objects

The entities in our model are *unique ids, tuples, values, relations, images,* and *links.*

> *Unique ids:* these are location-independent unique names for objects that low level clients can refer to. By low level clients we mean only those software functions needed to implement the view level effectively. This of course includes the compiler, for example. Tuple types, tuples, relations, and images have unique ids. At this moment we intend that unique ids are not available to user programs (that is view levels and above). If they were available, we would have to deal in a more general way with the problem of garbage collection.
>
> *Tuple types:* a tuple type is like a Mesa record type. It defines the form of a tuple by specifying a set of *fields*, each of which has a name and a type. The name is a string; the type specifies what kind of value appears in that field in an actual tuple. A *link* is a special type of field containing a tid (see below).
>
> *Tuples:* A tuple is like a Mesa record: it is an instance of a tuple type, in which each field contains actual data. The unique id of a tuple is called a *tid.* Tuples differ from Mesa records in that as the data base evolves we expect fields to be added and/or deleted. The view mechanism will assure that this will not cause type incompatibilities with application programs. Polymorphic facilities or partially specified types would certainly help make

tuples be a legitimate Cedar data type.

Value types: a value type is like any of the following Mesa types: interval, enumerated, string, array. Unlike Mesa, strings and arrays can vary in length after creation. Value types also include some form of reference to external files (e.g. a Juniper unique file identifier); if the Juniper file system were typed, file reference value types would include the file type.

Values: from the client's point of view, values stored in the data base look just like the corresponding Mesa objects.

Relations: A relation is a collection of tuples, all of which have the same type. Every existing tuple is in exactly one relation; a default relation exists for every tuple type, but once a tuple has been created, it may not be moved to another relation. A relation has a text name.

Images: An image is an auxiliary structure which orders a subset of the tuples of a single relation on some key, allowing fast determination of all the tuples whose keys fall in a given range. The ordering of the tuples is determined by some user defined procedure, the simplest case being a lexicographical ordering on some field. Images are implemented as B-trees.

Links: a link is a logical entity which connects a set of *source* tuples S in one relation with a set of *target* tuples T in the same or another relation. The implementation of links varies according to whether the link is uni- or bi-directional, and according to whether S and/or T is at most a singleton. We know that at least the following four kinds of links are useful:

1) Uni-directional, T a singleton. These might be useful to connect tuples of a relation so that scanning would be efficient.

2) Bi-directional, S a singleton. This is particularly useful in implementing a hierarchical data model.

3) Unrestricted bi-directional. We can represent joins efficiently with this structure.

4) Ordered links. This is a bi-directional link implemented by an array containing the tid(s) of S followed by the tids of T. This might be useful in representing a hierarchical data model in which operations such as go to the i'th child are allowed.

Transactions

Transaction mechanisms control crash recovery and deadlock restarting: a data base transaction has the property that it is guaranteed to either complete (in which case all its writes have been done) or abort (in which case none of its writes have been done). We will provide this facility to clients, built on the Juniper transaction mechanism. Clients must decide when to tell the system that a transaction is complete, and must be prepared to deal with unsolicited notification that the current transaction has been aborted because of system failure or lock conflict. We do not intend to support sophisticated mechanisms such as predicate locks and the like.

The client transaction mechanism will be implemented by using checkpoints within a single very long Juniper transaction.

II.1 Pragmatics

Storage organization

Strings may be stored either as byte strings or as "atoms" in the Lisp or Cedar sense. The latter representation conserves storage space and comparison time (at least for equality comparison) at the expense of conversion time and accessing a large table. The former representation is the default; the client may specify the latter for a particular field of a particular tuple type, or for a particular field of a particular relation. The choice of representation does not affect the semantics.

The storage format for links depends on whether the source and target are singletons; if not, either a chained structure or an indexed array may be used. The choice of chained or indexed structure may be left to the client.

Juniper

Most of the transaction and concurrency mechanism supported by Juniper is quite appropriate for the basic storage system for the data base. Juniper's current locking algorithms for supporting transactions are described in the memo [Ivy]<Israel>InterimLocks.bravo. Juniper currently locks entire files; in the near future it will be extended to offer page level locks, and possibly even byte range locks if we ask for them. If Juniper does not offer byte range locks, we will implement them using page locks, accepting the coarser grain and higher rate of conflict.

Distribution of data and function

We find it natural to use Juniper for the storage system. We believe it is not advantageous to place the low level data base interface on Juniper, because

1) there is no protection against the low level system destroying Juniper;

2) since Juniper supports relatively long transactions (tens of minutes), the overhead of transferring pages between Juniper and another machine implementing the low level interface can be substantially decreased by caching a large number of pages.

Our considerations suggest that there is no need in the Cedar environment for any intermediate servers between Juniper and clients. If Juniper's relatively crude protection machinery proves inadequate for some future application, or if an application arises with a very high access rate for a particular collection of shared data, we recommend placing an *application* server between the client and the actual data. (The White Pages might be such an application.) Otherwise, in general, all of the database software will run in the client machine.

Additional (but less definitive) thoughts on this subject can be found in DS-highlow.memo and DS-transaction.memo.

Caching

Pragmatic information supplied by the client can affect several key aspects of the data cache held on the client's machine:

What size unit to fetch (individual tuples, entire pages, entire relations, parts or wholes of images, etc.);

How much to retain out of a fetched unit;

How to determine when to release something from the cache (other than when required by a locking conflict);

When to prefetch information that will be needed by an access operation in the near future.

The details of how this information will be supplied are not specified, although we intend that the client only need deal with the same concepts as are already present in the high level interface. We believe it is necessary to give the client a great deal of pragmatic control in this area because of the wide variation in application requirements.

Additional thoughts on this subject (although based on a former system structure that included a server between the client and Juniper) may be found in DS-transaction.memo.

Locking

Juniper's consistency mechanism gives clients a simple but powerful mechanism for locking tuples. For other purposes, however, we may want to lock larger or more complex objects: relations, fields, subranges, images, or links. One way to accomplish this is always to check a unique flag associated with these objects whenever we access them: writing a flag will set a write lock in Juniper, which assures the client undisturbed access to the object.

III. View Level Functionality and Architecture

The intermediate level of the data base system will be composed of a general version of data base *views.* Views provide a way to describe data structures in terms of others through multiple levels of mapping. They serve several functions:

1. Data independence and Data abstraction - programs which access the datbase through views need not be changed when the actual storage structure is changed, although thy may need to be recompiled. Only relevant view definitions need to be altered. Details of implementation need not be the direct concern of program developers.

2. Support of various data models - It appears that within the lab, various groups will want to access data in a relational or network form. The same data should be available in both ways to facilitate sharing.

3. Performance support (system pragmatics) - Views, as the client sensible means of describing the data, are the means by which the system can be told how to organize, group and cache the data. The support of indexes is merged with views. Links, a stylized form of pointers, are also supported.

4. Computed fields and triggers - These functions are directly supported in our concept of views.

III.1 Interface definition

A view is defined by some declarative information together with a procedural specification, as indicated below.

View = =

1.0 *Import List* {Specifies those *views* used by this one to specify its value. The imported views are also typically modified when the view being declared is modified. If the Import List is empty, then this view is a base relation.}

2.0 *Export List* {Specifies those *operations* that this view supplies to the outside world; i.e. those which may be used by client programs as well as by other views in defining themselves in terms of this one.}

{The operations that are to be supplied should include the ability to retrieve and store a unique

tuple, to sequence through the tuples in a view in a particular order, and to retrieve a set of tuples (therefore defining a new relation) according to some simple retrieval criteria. It should also be possible to find the tuples, in another relation, which have been linked with a given one by a *prejoin* (see below). This ability supports networks and hierarchical models of data.}

{It is possible to program the standard relational algebraic operators in terms of operators such as these. In such an approach however, it would then be required of the compiler that the inner loops be significantly optimized. An alternative, which puts more load on the view definer, would be to have him provide an operator which returned large pieces of the relation at a time, in such a format that the join operation, implemented in microcode, ran very quickly without significant action by the compiler.}

3.0 *Pragmatics* {Here the view definer gives specifications that may be used by the system to take actions which can aid performance. *Nothing* in this section carries semantics; that is, the query and view definition language (they are the same) does not refer to constructs discussed here.}

3.1 Images: {An image is an ordering on all the tuples in a single relation. There may be multiple images defined for a given relation, or none at all. If only one is defined, it is used as the physical storage order. If more than one is defined, one can be specified as preferred, and used for storage contiguity, in which case updates cause reorganization. The system will use the image declarations to set up b-trees. They will be maintained dynamically in the face of updates (phew, this could be a big job!)}

Image X: (domain-list) {The given lexicographic ordering of the domains defines the sort order of the relation. Leftmost domains are most significant.}

3.2 Stored/Computed {This flag indicates whether the view is to be actually stored in the database, or whether it will be computed on the fly when accessed.}

3.3 Prejoins {It is possible for several of the relations listed in the *Import list* to be prejoined. The effect of doing so is to have the system construct the links indicating which tuples in each relation are matched with tuples in the other relation. This facility helps support network and hierarchical models of data efficiently. Notice that the number of links per tuple is variable. In an argument to a prejoin specification, one can say whether the system is to doubly link (i.e. install links in both directions) or not.}

3.4 Caches {In this part of the view specification, certain pragmatic information may be given, such as the name of an image. It may also be able to give a set of queries which partition the view, for those relations which are huge and cannot be cached as a unit. The system will use these specifications as a default cacheing guide. The action to be taken, i.e. what the system chooses to cache based on this information, is not yet specified.}

3.5 Sharing and Protection {It is possible to state the level of synchronization to be supported for this view. Access control may also be considered, although certainly not for a first implementation.}

Note that, while each of the above pragmatic specifications may be given as part of the view declaration, it is also reasonable for it to be possible for all of them to be given dynamically as hints by a running program.

4.0 View Body {The body of a view can be given in one of two "languages": a Mesa subset, and a limited procedural language consisting mostly of the usual relational algebraic operators, augmented by special purpose functions that are found to be useful in the Cedar environment. Some view definitions may be quite limited; permitting retrieval only, for example. Others may not only be complete, but may have logically redundant operations included for the sake of performance; retrieval of individual tuples as well as the entire relation, for instance. In any case, the actual definition has two parts:}

4.1 Results Section {This section defines the value of the view as seen by other views or clients which use it. It is the usual view definition, and must include a function that returns the entire relation being defined. This is the "pulling" part of a view. The Results Section may also include additional exported operations on the view, as listed in the Export List.

If the view is a base relation, then the actual data must be supplied. The structure of the data is given in part II of this document. The syntax is unspecified, but presumably will be nearly a subset of Mesa.}

4.2 Effects Section {This section provides the inverse of the Results Section; the "pushing" part of the definition. It is a set of exported operators that, when invoked, cause modification to the view. If the view is to be modifiable at all, there should be included in this section at least the support for the operations included in the relational algebra. The basic one is assignment, but more global operators may be desirable.}

End View

The view level provides most of the objects supported by the low level as user-accessible objects. It also makes the view definitions themselves available; clients can, under appropriate constraints, create and alter them.

We repeat here that **user-visible objects** through views are not the same as the objects visible at the low level. Unique ids are not visible. Links and images carry no semantics. They exist merely as hints for efficiency.

Data Consistency Control

We do not propose to support any integrity mechanism at the item level, aside from type checking. However, we will support two consistency levels similar to those of System R. Each client transaction specifies which consistency level it wants the system to guarantee:

> Level 1 only guarantees that other transactions appear atomic with respect to this one, i.e. writes by other transactions can affect data being read by this one at any time, but such writes all appear to happen instantaneously for a single other transaction. This is implemented by breaking the level 1 client's read locks in the case of conflict, but not aborting the transaction.
>
> Level 2 guarantees that no other transaction will write on data read by this transaction. This is the normal Juniper guarantee. The client must, however, be prepared to restart a level 2 transaction if it is aborted because of a write by another client's transaction.

Thus the significant difference between the two is that the second breaks locks when conflicts occur and the first does not. For more discussion of these points, see the DS-transaction memo and the System R paper.

III.2 Pragmatics

Storage organization

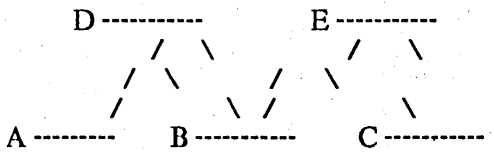
The client may specify a variety of "static" pragmatic information in the view definition which permanently affects the way in which data are stored in the data base. Examples of such operations and their corresponding effect on the storage organization include:

> A retrieval using a particular field as a key, which would cause the system to maintain an image of the relation using that field;
>
> A join, which would cause the system to install links and possibly co-locate corresponding tuples of the two relations on the same pages.

The client may also specify "dynamic" pragmatics with an individual query, which the compiler may take into consideration in deciding how to execute the query most efficiently. An example would be a statement of the cardinality of some intermediate result (join or intersection) relative to its operands.

Computed Fields and Triggers

The view facility has been designed to provide the functions of computed fields ("pulling") and triggers ("pushing") as natural consequences of their definition. The Results Section of the Interface Definition (see III.1) provides the computed field definition directly. Triggers are supported by the fact that view definitions may overlap, and that effects of an update to a view on the underlying relations used in the view can be specified. That is, in the following diagram, A, B, C, D,and E are views, with D defined in terms of A and B, and E is defined in terms of B and C:



a change to D that affects B can be subsequently seen by a reference to E. Some notification mechanism will be needed to make this all work; to propagate all the necessary changes, especially when some client updates a lower level view, such as A, B or C, directly.

The ability to update a lower level view when a higher level one has been defined on top will be an option, controllable by the definer, so that if it is not needed, the definer will not have to specify the change propagation machinery.

Compilation

A key idea in our proposal is the ability to compile an access operation (query or update) and have the compiled form remain valid for a long time. This allows us to provide great generality at the high level interface and still obtain good performance.

We extend the usual notion of queries and updates to include operations that take parameters. This is a natural extension since the compiled form of an access operation is a procedure. Clients who access the system directly at the view level do so by compiling a set of access procedures which fetch and retrieve individual tuples or fields, in which a key value is a parameter.

The use of compile-through to support a high-level interface efficiently is described in DS-highlow.memo.

The ability to compile access procedures, together with changes to the long lived data being accessed, forces the recognition that code may have to be recompiled even though the source code involved may not have been changed at all. Until an implementation plan is developed, it is not clear exactly which changes to a view will require recompilation of procedures which use the view, but there are likely to be many.

Pragmatic Specification in Interface Definition

The specification mechanism for views is intended as the only way by which the client can advise the system with respect to resource management, both static (i.e. influencing the means by which the system stores data over a lifetime longer than program invocation) and dynamic. Thus the language in which the running program will advise the system regarding caching and other matters is also patterned after the view specification facility. The *pragmatics* section of the view definition **cannot** be used in the expression of an algorithm.

Currently, the actions that the system is to be prepared to take as a result of client advice include: B-tree maintenance on various fields of a relation (including creation and deletion), link support between the tuples of two relations, storage of tuples of a given relation in a specified physical order, and various caching decisions.

View Implementation Language

Two languages are envisioned for the actual specification of the Results and Effects Sections of views. The one of choice will be basically a simple relational algebraic language, suitably extended with additional operators that are found to be of use in Cedar. The escape will be to permit the use of Mesa code. Both implementation languages, Mesa and the restricted view definition facility, will require significant optimization support. For Mesa, the definitions of views which are used in a client program may be treated as macros, and optimized by the compiler. The non-procedural specification should permit more extensive improvement.

The view facility is designed with the ultimate loophole of Mesa code so that it should be possible to accomodate nearly any user action. Clients are encouraged to define their views in the more restricted language provided however, as it affords the system more opportunity for analysis and optimization.

Interaction with the rest of Cedar & Requirements for Cedar Language Design

The view mechanism is intended to be the only way by which most non-privileged software can access stored data. Therefore it must be highly efficient. A great deal of optimization will be needed. The goal is for all permanent Cedar data storage to employ the data base storage facility proposed here. The major benefits are of two types: sharing and interchange of information is greatly eased, and the work of developing data manipulation facilities is reduced for many people, who would otherwise develop different versions of similar functions.

Relation of Views and Abstract Types

There is a close relation between usual ideas of abstract types and Cedar views. The differences result primarily from sharing, visibility, and the nature of the specifications which are encouraged. It is perfectly reasonable, for example, to define a view D that uses existing views A and B, and also to define view E that uses existing views B and C. The result of doing so is that changes to D cause changes to E. This kind of side effecting is desired in data bases but often considered inappropriate for abstract types. It is also reasonable for A, B, and C to be still visible to clients for either reference or modification, even though X and Y are defined. Lastly, since views have a long lifetime and are used for pragmatics advice to the system, additional statements regarding storage disciplines can be made about them which are not normally said about the definition of abstract types.

IV. Conclusion

We are proposing a system to serve long range CSL database needs in a wide variety of applications. Since relatively early development stages of Cedar, such as consistent compilation and program library, require database support, we propose a staged implementation.

We intend that even the first version will include caching and transactions. At first only the most simple concurrency control and locking provided by Juniper will be available. Simple compilation, to S-expressions which can be interpreted, or no compilation will be present at first.

Appendix I. An Example Use of the Cedar Database Facility

Assume the existence of a data base containing the equivalent of the CSL Archives, and another set of data bases containing messages received by each member of CSL. Note that even though each of these data bases contains conceptually similar information (i.e. each entry has fields like Author, Date, Subject, To, Text, etc.), their data may be stored in quite different ways:

> Documents are typically stored as Bravo or Tex files; the fields are implicit in the formatting of the document.
>
> Messages are typically stored as Laurel files; the fields are explicit in the message header.
>
> In the future, documents may be stored as a collection of field values, together with a pointer to a form to be used for creating their visible representation.

Thus, any program that wants to treat these different forms identically must use the view mechanism.

As an application, consider the situation when a person has just returned from vacation and wants to read through all documents and messages of interest, selected by subject. Here is a mock Mesa program to do this:

```
FindDocuments: PROCEDURE[startingDate: Date, subject: STRING]
    = BEGIN
    allDocs: SET OF AbstractDocument =
        FIND doc IN UNION[PublicDocuments, MyMessages] SUCHTHAT
            doc.date >= startingDate AND Occurs[subject, doc.subject];
    doc: AbstractDocument;
    FOR doc IN allDocs DO
        WriteString[DateToString[doc.date]];
        WriteString[doc.author];
        WriteString[doc.id];
        WriteChar[CR];
        ENDLOOP;
    END;
```

Appendix II. Potential Cedar Database Clients

The following chart summarizes the Cedar Database group's survey of potential database clients.

Dimension	Program Store	CAD programs	Page Imaging	AI (KRL)
Size (in tiniest items known by the DB)	10^4	10^6	10^4	10^5
Generality (desirable spec features)	low/medium	reas retrieval queries, thats all	low	triggers, computed filds, simple lookups
Performance (time for one operation)	modest	high? (prog access)	low; high for data transfer	high
Protection (from access by others)	low (ie, simple)	none	low	low
Compile-thru of DB repres	not needed	not needed	not needed	not needed
Type Integration (into Mesa)	(long range)	yes (would be nice)	not an issue	if needed for perfm
Data Model "most natural" for client (?)	relnl for symm questions, but also trans closure, etc	hierarchical? (don't care)	hierarchical? (don't care)	network perf, relnl model ok
Distributedness (of the data, and the users)	shared+local w/interlock; "fat wire"	simple locks at chip level; else no	consistent compln and distribution	centralized, but multiple users nice
User Interface (needed here?)	browser (by others)	no, they'll do it	no strong requirement	no
Prgrm Interface (level needed)	yes- minimally for compiler, editor, etc	yes	send filter to data?	yes- direct refs (RSI) (queries dynamic)
Phys integrity (losing bits)	crucial	crucial	moderate, crucial for user-inputted	no
Semantic integ (assertions on data)	important	wonderful if sophisticated	"distributed" docs	yes, nice
Transn volume (access and update)	<100/day(/user)	10^4/hr simple? 10/hr query level	few/hr	10^5/hr (direct)

Dimension	Masterscope	Personal DB asst	PARC Library	3rd Floor (OIS)
Size (in tiniest items known by the DB)	10^5 max	10^4	10^5	10^5 ?
Generality (desirable spec features)	tran clos most, general retrieval	not necessary, but nice (eg views)	text search, save queries/reslts	canned modules: triggers, computed fields nice
Performance (time for one operation)	modest (high at lowlevel?)	modest (high at lowlevel)	low	modest (high for batch)
Protection (of access)	none	low (user + system)	low	high: salaries, etc. (phys sep'n OK?)
Compile-thru of DB repres	not needed	not needed	not needed	not needed
Type Integration (into Mesa)	(long range)	no	no	modest, some nice
Data Model "most natural" for client (?)	network perf, relnl model ok	network perf, relnl model ok	relational	hierarchical or relational
Distributedness (of the data, and the users)	shared+local	shared+local	link to extrnal DB (muliple users)	no; shared
User Interface (needed here?)	no, provided	no, provided	yes. limited resources	yes. limited resources
Program Interface (level needed)	yes, low level	yes, low level	straightforward	straightforward
Physical integrity (losing bits)	no, all derived information	crucial	crucial if no other backup	crucial (they'll have our heads if..!)
Semantic integrity (assertions on data)	no	not necessary, would be nice	yes, but special kind: formatting	important, but may be built into canned mods
Trans'n volume (access & update)	10/hr(/user), higher at lowlevel	10^3/hr lowlevel, 10/hr update	10/hr now; but high w/user accs	100/hr peak

Date: 5 Jun 1979 9:44 am (Tuesday)
From: Satterthwaite
Subject: Re: Pointing vs. Naming, etc.
In-reply-to: Your message of 31 May 1979 3:36 pm (Thursday)
To: Horning
cc: Cedar-Lang↑, Lampson, Owicki

I am in general agreement with your analysis of pointing vs. naming (although I am somwhat dismayed by the number of different flavors of pointers that are currently being invented).

Mesa's OPEN comes close to binding a name to an internal reference constant, but it currently fails to serve your purposes for two reasons.

The first (less interesting and, except for compatibility issues, less substantial) difference is that OPEN was originally designed as a mechanism for augmenting Algol-like visibility rules of Mesa. In implementation terms, it was intended to add a programmer-specified display entry to the local environment. Since Mesa uses records to model frames, argument lists, etc., OPEN was defined to take a record OR a pointer to a record (with automatic dereferencing in the latter case). Thus OPEN x: p and OPEN x: p↑ are equivalent if p is a pointer to a record; the former does NOT bind x to a reference to p.

The second difference is that the binding is by name. The expression designating the opened variable is not evaluated once and the value preserved; it is recomputed each time the variable is mentioned. Once-only evaluation doesn't work in the presence of a relocating allocator (except, perhaps, one hidden and built into the implementation). If a finger is dereferenced or a relative pointer is converted to an absolute one, the reference is bound prematurely and, in general, incorrectly. Binding by name avoids this problem but has others; most notably, it allows the logical identity of the OPENed object to change within the scope of the OPEN. (This problem is especially severe in a discriminating WITH clause, which is just a thinly disguised OPEN.) Resolving these difficulties is one of the goals of our search for a good characterization of a "generalized pointer" (and now "generalized reference" also). There should be some way of separating the computation defining the reference from the computation doing the dereferencing; e.g., OPEN should indeed compute the value of a finger just once, but that value should be dereferenced one extra level on each use. At one point, the Mesa Language Working Group thought that a language extension supporting all this was almost within our grasp. (Maybe application-specific relocating allocators are irrelevant for Cedar; they seem to be necessary for product software running on small machines, even when the virtual address space is large.)

By the way, I agree that the "empty" name feature for OPEN is a mistake and should be eliminated just on reliability grounds. Be aware, however, that it has some ardent defenders. Without it, for example, splitting a module requires rewriting all references to those globals in one part from the other. (This probably says more about the overloading of modules than about OPEN, but ...).

Date: 7 Jun 1979 11:32 am (Thursday)
From: Horning
Subject: 6-6-79 Minutes, cont'd: Variant References?
To: Cedar-Lang↑, Lampson

After the secretary for the day (Morris) left, we continued our discussion of references, and obtained (at least the illusion of) further insights.

We were dealing with the tension between wanting to always keep pointers, references, and internal references properly sorted out--which we had assumed meant carrying this as part of the type of the "address-containing" variable--and the desire to be able to write rather general routines that were independent of the source of their arguments.

We were also getting rather discouraged at the number of different kinds of "address-containing" types that were being proposed: pointer, long pointer, reference, internal reference, var parameter, opened variable, handle, universal pointer, etc.

There was also a question of whether short pointers had a future in Cedar.

The primary insight that seemed to emerge is that these variations and discrimination requirements have very much the flavor of associated with a variant record type. We even noticed that 28-bit addresses leave a few bits available for a tag. Consider the following (conceptual) variant record type:

```
Reference: TYPE = MACHINE DEPENDENT RECORD [
    Address: LONG POINTER,
    SELECT tag: * FROM
        uncounted, counted => NULL,
        internal => [offset: POINTER],
        universal => [type: TypeIndication],
```

```
    abstract  =>  [concreteType:  TypeIndication],
    ...
ENDCASE ];
```

Undiscriminated Reference variables not on the stack can simply generate transactions on the collector's queue--the collector can inspect the tags and discard transactions dealing with uncounted references. The stack scanner should probably do the discrimination.

Whether @ generates a counted or internal (e.g., to a stack frame) reference depends on the sort of variable to which it is being applied. However, the knowledge of the correct variant is always available (I think) at the point where @ is being applied (if not, we could probably find restrictions to make it available). I hesitate to say this, but it appears that this makes @ completely safe. Free and Loopholing to uncounted Reference (or short pointer) would now appear to be the principal unsafe pointer operations. Have I overlooked something?

This allows us to declare uncounted Reference variables, universal Reference variables, etc., just as with other variant record types, and compile-time checking will keep them straight for us.

It will also be possible to assign an uncounted Reference or a counted Reference to a Reference variable; the tag will be created automatically, as for any other variant record type. However, in order to make an assignment in the other direction, it will be necessary to use a discrimination, to ensure the type-safety of the assignment:

```
WITH discrimRef: refVar SELECT FROM
    uncounted  =>  pointerVar ← discrimRef;
    ENDCASE  =>  SIGNAL  ReferenceToPointer;
```

The effect of VAR parameters is now very nearly simulated by giving parameters Reference types. We can live for some time without the syntactic nicety of the implicit @'s and ↑'s.

OK, shoot it full of holes.

Jim H.

Date: 7 JUN 1979 2248-PDT
From: DEUTSCH
Subject: Variant References
To: horning
cc: Cedar Language Features Design Committee:, lampson

The only problem I see (but it is a serious one, in my opinion) is that an undiscriminated Reference takes 3 words. It is not clear to me that we are willing to pay this price in storage and time in every place that MIGHT contain an internal Reference. The proposal for type-carrying pointers that I laid out in the "type aspirations" paper (and am repeating in the forthcoming delayedbinding3 memo) carefully avoids any necessity for actually carrying a type word along with a pointer.

I also think we may want to split the tag into one tag for allocation {counted, uncounted, internal} and one for dynamic typing (delayedbinding3 proposes {quantumMap, previousWord, hashLink}).

Everything else looks good.

Notice that we have started down the garden path towards making Mesa into a tagged 32-bit architecture, which none of our hardware supports very effectively. ------

Date: 8 Jun 1979 9:51 am (Friday)
From: Horning
Subject: Re: Variant References
In-reply-to: Your message of 7 JUN 1979 2248-PDT
To: DEUTSCH
cc: Cedar-Lang↑, Lampson

Peter,

Perhaps I didn't make it sufficiently clear that this was intended as the SEMANTICS of references, not necessarily their implementation. I agree that we need to think very carefully about how to map frequently-used types onto the machine architecture, and hope that nothing I proposed is in conflict with that ideal.

I expect that, except for the case of "virtual VAR parameters," the use of undiscriminated Reference will be considerably less frequent than use of the various discriminated forms. I also propose to produce a short note demonstrating that short addresses fit nicely into this conceptual scheme.

Yes, I probably should have used about three tag fields for conceptually distinct axes of variation; I didn't just because it wasn't present Mesa, which isn't a good enough reason.

Jim H.

Date: 8 Jun 1979 10:35 am (Friday)
From: Horning
Subject: Short Addresses: Back up the garden path
To: Cedar-Lang↑
cc: Lampson

Although Peter is worried (or secretly pleased?) that our recent attempts to make some conceptual unifications of the various "address-containing" types that have been proposed for Cedar Mesa recently are leading us a long way down the garden path towards a 32-bit tagged architecture, it seems to me that we can keep the frosting while eating the cake as we walk back up the garden path. (I just can't resist metaphors if the mixing is sufficiently incongruous!)

The basic idea is that we introduce a new type constructor, RELATIVE. For
T: TYPE = rConst RELATIVE RType; rConst may be any reference constant (MDS is implied if this operand is omitted), RType may be any (absolute?) reference type.

The semantics of T are derived from those of RType--essentially by coercing the T value to RType and then performing the RType operation (except for T-assignment, of course). The representation can be more compact (i.e., 16 bits), since rConst.address is added on each reference. Assignment of an RType value (e.g., the result of @) to a T variable may require a run-time range check, although the compiler can generally notice when "cancelling" pointer arithmetic is implied (i.e., short addresses can stay short).

The requirement that rConst be constant ensures against the base of a short address changing out from underneath it. Making rConst part of the type prevents any rConst RELATIVE references from outliving it (provided we have correctly implemented persistent closures). It also keeps us from confusing relative references with different bases. Note, however, that the TYPE of the base reference is irrelevant to the relative reference, which has its own type.

Note that present Mesa POINTER TO T' maps pretty straightforwardly to MDS RELATIVE uncounted REFERENCE TO T'. This seems to give the right conceptual basis for continuing to allow present usage while maintaining safety.

Jim H.

Inter-Office Memorandum

To	Cedar Language Committee	Date	June 12, 1979
From	Bobrow & Deutsch	Location	Palo Alto
Subject	Language changes for "objects" in Mesa	Organization	CSL

XEROX

Filed on: [Ivy]<CedarDocs>lang>BD2-objects.memo

This memo describes our proposal for "objects" in Mesa, based on the current Mesa ideas of interface and module instance. We start with a short discussion of four properties of objects which must be known to clients of such objects, and recast different uses of objects in these terms, including Butler's proposal. We then make our specific proposal.

Properties of Objects

To use an object of a specific type, a client needs four types of information:

1) *Syntactically legal protocols (at compile time):* What are the legal operation names and arguments.

2) *Storage (at compile time):* How much storage to allocate for an instance of an object; if a garbage collector is to be able to trace the object, then additional information to interpret this storage is necessary.

3) *Method lookup (at run time):* How to find the code for the operation given the particular instance of the object representation stored in the client, and how to pass the storage representation to the method.

4) *Consistency checking (at run time):* How to ensure that the piece of code found actually agrees with the legal protocol ensured by 1.

Examples

Legal Protocols: Lisp record accesses (which can be general operations) are checked for legality by ensuring that the field name is defined for one and only one record. Message protocols in Smalltalk are unchecked at compile time. Humus objects use a Defs module specification of procedure types, as does Butler's proposal and our object proposal.

Storage: Lisp stores all objects as typed pointers, so storage size is fixed and so is interpretation by the garbage collector. Smalltalk also uses a uniform representation scheme for all objects as did Humus. Butler's scheme allows variation in this dimension, and is specified by the declaration of type in the Defs module. We are proposing a standard representation of POINTER TO FRAME.

Method Access: Lisp uses the property list or function cell of the Litatom to find the method in a standard way. For records, this is done at compile time. Lookup uses a system wide procedure and is independent of the object instance that the method is going to be applied to. In Smalltalk, a message name is constructed out of the message pieces in a standard way. The lookup method uses a system wide procedure for finding a message name in a dictionary associated with the particular

object. In Humus, an index into a record of procedures is compiled into the code, where the index number is computed by the compiler on the basis of information in the Defs module.

In Butler's scheme, significant variation is allowed. The only property that Butler insists on is that "the compiler generate the same byte codes to get to the object dependent part of the code." While this variation may be elegant in principle, it seems to lead into research areas about how to ensure that implementing modules actually generate the same code, and we felt it better to punt that issue right now. Our scheme uses a scheme similar to Humus, in which the client indexes into a known record.

Consistency checking: In typed record accesses, Lisp explicitly checks the type of the object at each access; in untyped record accesses, it simply believes the programmer. In Smalltalk, the only consistency check is whether the message name actually appears in the dictionary of the specific object. In Humus, the system is forced to believe the programmer (through the use of a necessary LOOPHOLE at object creation time). In both Butler's proposal and ours, the Mesa compiler will ensure the type safeness of the transaction.

Specific proposal

We define a new kind of DEFINITIONS module introduced by the keywords OBJECT DEFINITIONS. The semantics of such a module differ from those of an ordinary DEFINITIONS module only in the following ways:

> If ObjDefs is an OBJECT DEFINITIONS module which is known to module ObjClient, then the type POINTER TO FRAME[ObjDefs] is available in ObjClient. Such a pointer may point to any global frame of any implementer of ObjDefs.
>
> If ObjClient IMPORTs an instance of ObjDefs named ObjInst, the effect is exactly the same as defining ObjInst as a variable of type POINTER TO FRAME[ObjDefs]. However, in contrast to the usual behavior for IMPORTed interfaces, there is no way for the Binder or Loader to create an instance to store into ObjInst, since there is no way to tell which implementation of ObjDefs is wanted. (Note that if ObjInst = ObjDefs, i.e. the IMPORTS statement does not provide an explicit name, then ObjClient cannot refer to the interface ObjDefs at all, only to the instance pointer. Perhaps this should be an error.)
>
> > The only reason for allowing IMPORTing of OBJECT DEFINITIONS at all is to allow clients to write code that will work regardless of whether the interface is defined by DEFINITIONS or OBJECT DEFINITIONS. Note that in the current language, the only way to create appropriate instances for IMPORTed OBJECT DEFINITIONS is to have another module that IMPORTs ObjClient directly, so it can store into the ObjInst variables.
> >
> > Part of the problem here is that the Binder cannot actually run the START code of the implementer, or supply parameters to it. If the Binder had those abilities, then it would be possible with little or no additional extension to C/Mesa to specify which implementer to instantiate for an IMPORTed object definition.
>
> If ObjImpl is a program module that EXPORTs ObjDefs, it must export *all* the PROCEDUREs, SIGNALs, and variables defined as interface elements in ObjDefs, and it must not EXPORT any other interface. The only exception to this is the use of the EXTENDS clause described below. The compiler arranges the global frame of ObjImpl so that the interface elements are allocated at locations computable from ObjDefs alone.
>
> The variable SELF is automatically defined in ObjImpl: its type is POINTER TO FRAME[ObjImpl] and it represents the frame of the currently executing instance of ObjImpl. This allows an implementer to get hold of its instance to pass as an argument, for example.

If ObjPtr is of type POINTER TO FRAME[ObjDefs], then for any interface element X in ObjDefs, ObjPtr.X is legal and is interpreted as though it were ImplPtr.X, where ImplPtr is of type POINTER TO FRAME[ObjImpl]. It does not matter which implementation ObjPtr actually points to, since all implementers of ObjDefs have their interface elements (global variables and signal codes, and entry points) laid out in the same way. In fact, it is legal to assign a POINTER TO FRAME[ObjImpl] to a POINTER TO FRAME[ObjDefs].

Creating new instances of objects is accomplished with a CREATE expression that is essentially equivalent to the current NEW and START constructs. CREATE ObjImpl[parameters] returns a POINTER TO FRAME[ObjImpl] just as NEW and START would; this is then usable as a POINTER TO FRAME[ObjDefs]. The difference between CREATE and NEW + START is that CREATE does not enter the frame into the global frame table (GFT). As a result, the usual representation for procedures in ObjImpl does not work, and we propose to use LONG POINTERS to get around this problem.

Long Pointers

Objects should not need to live within the same MDS, and should be designated by LONG POINTERs. Conceptually there is no problem with a LONG POINTER TO FRAME. Of course the system procedures that manipulate frames (like NEW and START) would have to be changed to take LONG POINTERs. Also, some changes would be entailed in the instruction set / virtual machine specification: G is not a LONG POINTER at present, and procedure invocations through POINTER TO FRAME are handled by actually constructing a procedure descriptor, which is limited to 16 bits.

In order for an object to construct a procedure descriptor for one of its own procedures, it must be able to provide the necessary information (LONG POINTER to SELF, and entry point number) which won't fit into 16 bits, even using indirection. We therefore propose defining a new LONG PROCEDURE type consisting of a LONG POINTER to a global frame and an entry point number. The bit representation would be arranged in such a way that one could inspect a cell and tell whether it was a PROCEDURE or the first word of a LONG PROCEDURE, so that the Call instructions which address the local or global frame would work equally well on ordinary and LONG descriptors. An instruction to lengthen a PROCEDURE would be very simple, similar to the Lengthen Pointer instruction, and would only need to be invoked when a procedure was needed in a non-call context. We advocate this solution for its relative simplicity and because it also solves another problem, namely the restriction on the number of global frames and the requirement that all global frames be accessible through an indexed table.

Known weaknesses in the proposal

Since an object is a global frame, it contains links for all procedures imported by its implementer. This is wasteful since every instance of the implementer (object) will presumably be bound the same. There are two ways to deal with this without further changes:

> Use the LINKS : CODE feature of the Binder, which moves all these links into the implementer's code segment.
>
> As much as possible, have implementers use objects for imported services rather than individual procedures. This reduces the overhead to one POINTER per object, rather than one PROCEDURE per procedure.

The overhead for a global frame is currently 4 words: a LONG POINTER to the code segment, a size code for the frame, some flag bits, and the GFI (index in GFT). Attempting to eliminate one of the overhead words, which might be possible since objects do not need a GFI, would probably be inconvenient since the instruction set and runtime system assume the present format, and would have to check at access time.

Circular Dependencies

In this proposal, it is not possible for one module to implement more than one object type (except for a chain of subtypes), because the entry point (operation) indices for every object type are assumed to start at 1. The only problem with this is that in order to implement interacting objects (e.g. a Node and an Arc of a graph) using separate Defs modules, there would be circular dependencies introduced between these modules. We propose adding a construct to allow the user to specify that a group of modules are interdependent and should be compiled together: this is the **CompileWith** clause in the syntax summary given below. The effect of this clause is that the compiler compiles all the mentioned modules simultaneously and gives them all the same time stamp. (Accidental omission of necessary interdependencies will be detected at bind time, of course.)

Multi-Module Object Implementations

Suppose OBJECT DEFINITIONS module DefsA defines operations P and Q. Then OBJECT DEFINITIONS module DefsB can define a new object type which provides operations P, Q, R, and S (with P and Q defined the same as in DefsA) by saying EXTENDS DefsA and just defining R and S. This has the following consequences:

> A module ImplA which EXPORTS DefsA must define P and Q. POINTER TO FRAME[ImplA] may be assigned to POINTER TO FRAME[DefsA].
>
> A module ImplB which EXPORTS DefsB must define R and S, and must say EXTENDS ImplA. POINTER TO FRAME[ImplB] may be assigned to POINTER TO FRAME[DefsB]. The compiler allocates a known slot (called the *superior* slot) in ImplB's global frame for a POINTER TO FRAME[ImplA]: if a client tries to assign a POINTER TO FRAME[DefsB] to a POINTER TO FRAME[DefsA], the compiler generates code to fetch the pointer from the superior slot.
>
> A module ClientB which uses objects of type DefsB can access the operations and variables defined in DefsB directly, but the compiler must produce code to follow the pointer in the superior slot wherever ClientB uses operations or variables defined in DefsA.
>
> If the division of implementation between ImplA and ImplB changes (e.g. ImplB now provides Q, rather than ImplA), all clients of DefsA or DefsB must be recompiled. However, if only DefsB is altered, neither ImplA nor clients of DefsA are affected.

This mechanism can be used both for multiple modules cooperating to implement a single object type (by not letting clients actually make use of DefsA or ImplA) and for subclasses (DefsB is a subclass of DefsA). Note that this approach even works for multiple superclasses -- all that is required is to have more than one superior slot, and a rule for handling the conflict that arises when more than one superclass defines the same operation name. We suggest for the moment that such a conflict be an error.

Summary of Syntax Changes

Our proposals require extending Mesa syntax in a very few areas: **CompilationUnit**, to handle circular compilation dependencies; **ModuleHead**, to handle EXTENDS; and **DefinitionTC**, to handle object definitions. The proposed revisions are as follows:

```
CompilationUnit ::=
        Directory
        CompileWith
        identifier : ModuleHead = GlobalAccess
        ModuleBody
```

```
CompileWith ::=
        empty | COMPILE WITH IdList ;

ModuleHead ::=
        ProgramTC ExtendsClause ImportsList ExportsList ShareList |
        DefinitionTC ExtendsClause ImportsList ShareList

ExtendsClause ::=
        empty | EXTENDS IdList ;          -- not legal with non-OBJECT
        DEFINITIONS

DefinitionTC ::=
        DEFINITIONS |
        DEFINITIONS LocksClause |
        OBJECT DEFINITIONS
```

In addition, as noted above, we propose adding LONG POINTER TO FRAME and LONG PROCEDURE, which are both allowed by the present syntax but disallowed semantically.

CSL Notebook Entry

PROPOSAL — Circulated for comment — PROPOSAL

To	CSL Programming Environment Interest Group	Date	June 28, 1979
From	Cedar Language Features Design Committee	Location	PARC/CSL, Palo Alto
Subject	Cedar Mesa Safe Subset Proposal	File	[Ivy]<CedarDocs>Lang>Safe.press

XEROX

Archive ID: yyCSL-xxx

Attributes: proposal, technical, Cedar, Mesa, Programming research

References: (All filed on [Ivy]<CedarDocs>)
"Unsafe at any speed?"—Jim Horning, Mar. 2, 1979
"Some Preliminary Thoughts"—Ed Satterthwaite, Feb. 14, 1979
"Minutes of 2-28-79 meeting"—Deutsch, Mar. 4, 1979
"Aspirations for run time types"—Deutsch, Bobrow, May 9, 1979
"Apr-May-Design.mail"
"June-Design.mail"
"Language changes for 'objects' in Mesa"—Bobrow and Deutsch, June 12, 1979
"Garbage Collection Safety"—D. G. Bobrow, S. Owicki, June 13, 1979
Invariants Proposal—Owicki, in preparation
Delayed Binding Proposal—Deutsch, in preparation
Objects/Exported Types Proposal—Morris, in preparation

Abstract: This document summarizes our current thinking on the "Safe Subset" of Mesa to be provided by Cedar. The motivation and required properties of the safe subset are described and proposals are advanced for specific changes to Mesa.

Introduction

Motivation

A desirable property of a high-level language is *implementation independence*, which means that the effects of each program are explicable in terms of the *language* (rather than its implementation)—even if the program is erroneous. Mesa comes rather close to meeting this goal (as evidenced by the fact that most Mesa programs can be debugged "at the Mesa level," without ever worrying about the format of stack frames or the details of storage management), but it does contain some "unsafe" features whose use can lead to messy implementation dependencies.

It would be desirable to reduce implementation dependencies in Cedar on general grounds. However, the decision to include facilities for automatic storage deallocation ("garbage collection") within Cedar Mesa makes a shift in this direction imperative. The collector can cause storage to be deallocated (permitting its subsequent reallocation and re-use) at times that are completely unpredictable from examination of the source program. A single programming error that smashes a pointer used by the collector can effectively bring the whole system down in a rubble of bits, from which it is difficult to reconstruct any evidence of the cause of the crash.

It has been one of the major goals of the Cedar Language Features Design Committee to design a subset of Mesa for which it would be safe to do automatic storage deallocation. To first approximation, the "safe subset" is that part of the language where even incorrect programs cannot interfere with the reliable operation of the collector.

We intend that the vast majority of programs will be written primarily in the safe subset. However, we recognize that it is not presently feasible to provide acceptably efficient substitutes for all uses of Mesa's unsafe features. Therefore, we propose instead to provide textual means for indicating that some regions of a program are *trusted.* This will inhibit compiler enforcement of the safety restrictions and simultaneously indicate that the programmer has assumed the additional responsibility of ensuring that these regions of the program cannot violate the integrity of the system.

This proposal has evolved after considerable discussion, taking into account reactions to the first reference cited above, which was circulated to CSL and the Programming Environment Interest distribution list. Comments and objections should be returned speedily if they are to affect the first version of the design of Cedar Mesa (informally called Mesa 5C1).

Invulnerability and Safety

It is an obviously desirable property of a programming system that no user programming error can reduce its world to a rubble of bits. We will call this property *invulnerability.* In general, it can be ensured only by maintaining the integrity of certain data structures known to the run-time system (e.g., dispatch vectors, core maps). Collectively, the properties that must be maintained to ensure invulnerability will be called the safety *invariants;* each portion of the system will be responsible for ensuring that they are not destroyed, and will be free to assume that the rest of the system does likewise.

Invulnerability is not a local property. If any part of the system fails to maintain the invariants, the entire system (including regions that are themselves correct) is potentially vulnerable. We use the term *safety* for the property that the invariants cannot be invalidated locally (even by incorrect programs). *Safe* program regions are portions of a system where safety is guaranteed by language-enforced restrictions. Even trusted regions are supposed to preserve the invariants, but the guarantee must be provided by the programmer, rather than by the language and its implementation. If a system consists entirely of safe regions (and the invariants hold initially), then by induction the program is invulnerable. However, an error in a trusted region could make even the safe regions vulnerable. Thus the safe/trusted boundary limits responsibility, but not vulnerability.

Only trusted regions may use unsafe language features. This terminology may appear somewhat paradoxical. However, "unsafe" refers to the fact that the system is vulnerable to errors in the use of these features, and "trusted," to the fact that confidence in the correct operation of the system must be based on trust that the region is free of such errors. ["Put all your eggs in one basket—and watch that basket!"—Puddin'head Wilson] By contrast, our confidence that safe regions will not crash the system is based on the knowledge that automatic checks have been applied.

Type Confusion

Mesa is a strongly typed language, which means that the types of variables are declared, and that the language inposes restrictions to keep values of one type from being accidentally interpreted as values of another. Because knowledge of the type structure of memory is so essential to the collector (it must locate and follow pointers in order to determine storage usage), it is particularly vulnerable to any operations that cause things in memory to be interpreted as having other than their declared types. Thus, much of the effort in designing the safe subset has gone into identifying all the features in the present Mesa language that allow type-checking to be circumvented (accidentally or deliberately) and designing safe replacements for important uses of those features.

LOOPHOLE, of course, is the most obvious means of defeating type-checking. It causes a safety problem if it allows mis-typing of some piece of memory (i.e., if the target type contains an address, such as a pointer or a procedure value); other uses will introduce implementation dependencies that do not threaten safety. We have formalized the distinction between these two sorts of breach of the type system, retaining LOOPHOLE in the unsafe language for arbitrary reinterpretation of the type of

a value, and introducing the safe language type transfer function PUN, which is semantically equivalent to LOOPHOLE, except that address-containing types are not allowed as targets.

Narrowing

This proposal introduces a number of new type distinctions into Mesa, frequently leading to a number of separate, but closely related types. It is often desirable to *coerce* a value of one of these types into a value of a related type. If it can be statically guaranteed that no information can be lost by such a conversion, we call it a *widening*. Widening will be performed automatically whenever demanded by context—this is analogous to lengthening a CARDINAL to a LONG CARDINAL. In general, conversion in the other direction will require a run-time check to ensure that information is not being lost. For each such *narrowing* described below we indicate the nature of the necessary check; if the check fails, the signal RangeError will be raised. To make the possibility of such failure explicit in the program text, the NARROW type converter must be applied whenever a narrowing is to be performed; it will also be applicable to present Mesa types (e.g., LONG CARDINAL).

References

Although Mesa pointers are nominally typed, they provide a rich source of opportunities for type confusion, including—but not limited to—the classical "dangling pointer" problem, where a pointer persists after the storage it refers to has been deallocated. We are introducing a new class of *reference* types to make explicit several distinctions important for safe garbage collection. Conceptually, REFERENCE TO is a type schema much like a variant record schema; we allow it to be particularized in the same way as bound variants. Certain bound variants correspond closely to traditional uses of pointers, and can generally be substituted for them. POINTER TO will be an abbreviation for one such variant of reference.

General Principles

System Invariants

Basically, the invariants that must be preserved are

> the integrity of the code and the various data structures maintained by the present run-time system for storage management, process scheduling, control links, etc.
>
> the integrity of the new data structures introduced for garbage collection (reference counts, transaction queues, symbol table information, etc.).
>
> the type integrity of all *address-containing (AC)* storage.
>
> the accessibility of all active AC storage to the collector by tracing links, starting from a known set of locations.

Further detail will be provided in the referenced Invariants Proposal.

Safe and Trusted Regions

Each block of a program will belong to a *safe* or a *trusted* region, with safe being the default. A trusted region may contain embedded safe regions, but not conversely. Our goal is to have safe regions predominate, and to make it easy for the reader to locate all trusted regions.

We have added the key words SAFE and TRUSTED to the language, to follow the BEGIN of a block. Unless a module body's block begins with TRUSTED, it is automatically safe. Unless a nested block in a trusted region begins with SAFE, it is automatically trusted. TRUSTED may not appear within a safe block.

Within a safe block, all the restrictions imposed on the safe subset will be enforced by the compiler (possibly generating some run-time checks for such things as array bounds). In trusted regions, the full language may be used, but the programmer is responsible for ensuring that he understands and preserves all the system invariants. Even in trusted regions it is likely that the bulk of most programs can be in the safe subset, and features outside the subset should be used only for good reasons.

Trusted Regions Providing Safe Interfaces

Although a safe module is not allowed to contain any trusted regions, it may *import* trusted modules. We distinguish between trusted modules and unsafe interfaces. A procedure that uses unsafe features to accomplish some safe operation (i.e., one that will not violate the safety invariants, even in the presence of errors in the calling program) has a *safe interface*, and such procedures may be called from within safe regions. Of course, the compiler cannot guarantee that a trusted procedure body actually provides a safe interface; this is the responsibility of the programmer who declares the interface to be safe.

The key word SAFE is also an access attribute. Preceding a procedure, port, process, or signal type constructor in a trusted module, it indicates a safe interface that may be invoked within safe regions. In its absence, the interfaces will be treated as unsafe, and invocation from safe regions will be prohibited. Thus, unsafe interfaces are private to trusted regions.

Safe regions are not allowed to raise unsafe signals, and trusted regions may only catch safe signals with safe catch phrases, i.e., those whose IdList is preceded by SAFE. [Note that this SAFE applies to the *interface*. If the statement in the catch phrase is to be a safe region, it must be written as a safe block.] An UNWIND, however, may be caught anywhere, because the region that raised it is sure to be terminated.

References and Other Address-Containing Types

AC Types

We need to impose restrictions on the creation and use of values of *address-containing (AC)* types. References (including pointers) are, of course AC, and are the primary concern of the collector. Several basic types in Mesa (string, procedure, frame, descriptor, signal, and port) implicitly contain addresses, and are AC. Arrays whose component type is AC, and records with any AC fields, are also AC.

Whenever an AC value is created all its address fields must be validly initialized (possibly to NIL) to avoid tracing or deallocating non-existent structures. In Cedar Mesa, all address fields of AC types that are not explicitly initialized will be implicitly initialized to NIL. (It is probable that in Mesa 6 type definitions will be allowed to include specifications of instance initializations; this will also override the implicit initialization to NIL.)

We must also guard against both "forgery" of AC values and the preservation of such values in locations unknown to the collector. The immediate implication is that we must prohibit LOOPHOLE and restrict UNSPECIFIED in the safe language.

As long as there is no mechanism for later converting them back to AC values, there is no

safety problem in allowing AC values (indeed, any values) to be interpreted as non-AC values, and we provide PUN as the safe replacement for LOOPHOLE.

In the safe language, values of type UNSPECIFIED do not conform to any AC type, although values of AC types may be coerced to UNSPECIFIED. REFERENCE TO ANY provides type-carrying references, which can safely be used to replace many uses of UNSPECIFIED.

Automatic Deallocation and Reference Counting

Cedar Mesa provides three kinds of storage:

Frame: This is storage that is implicitly allocated at procedure activation or module instantiation to hold variables declared in the corresponding scope. It is also implicitly deallocated—normally on scope exit, although other cases will be discussed below.

Heap: This is storage that is explicitly allocated by NEW, and deallocated by (unsafe) FREE statements.

Collectible: This is storage that is explicitly allocated by NEW, but which is implicitly deallocated after there are no more accessible references to it.

Present Mesa provides the first two kinds. Freeing the programmer from the necessity of explicitly deallocating storage (and the possibility of making mistakes in such deallocation) has been one of the major goals in the design of Cedar Mesa.

Although the programmer should not be concerned with implementation details of collectible storage, some understanding of the intended implementation will make it easier to understand the restrictions imposed on the safe subset.

The basic approach to garbage collection is the same as that advocated by Bobrow and Deutsch in their Sept. 1976 *CACM* article. Responsibility for reclamation is divided between two collectors:

Full collector: This program traces the entire active data structure and generates a compacted, but otherwise equivalent, copy. It must discard inaccessible circular structures and cope with pathological cases, but it is allowed to be slow, and will only be invoked when the system is otherwise quiescent, so it can be written knowing that the data structure will not be changed by other processes while it is running.

Incremental collector: This program is intended to reclaim most of the inaccessible storage at much more frequent intervals. It need not reclaim all the inaccessible storage (as long as it "usually" reclaims "most" of it), but must be able to work incrementally with low overhead while other processes are active in the system.

The incremental part of the B-D scheme requires the same information as simple reference-counting schemes. Conceptually, each object in storage has an associated count of the number of references currently pointing to it; objects may be deallocated at any time after their reference count reaches zero. However, the B-D scheme gives special treatment to commonly-occurring cases, greatly reducing the cost of the record-keeping. We expect its efficiency to be good enough that collectible storage will replace most present uses of heap storage.

Variants of Reference Types

There will be four "dimensions" of variability for reference types; in general, greater efficiency and earlier checking can be obtained by binding some or all of them in declarations. We discuss each in turn.

Packing = {unzoned, zone-relative}: By default, references are unzoned, and contain full machine addresses (28 bits). However, within the scope of a *zone*, many references can be represented with a

smaller number of bits (e.g., 16) as an offset from the zone's origin. Zone "variables" must have type ZONE and fixed initialization (i.e., be constant); they may be exported to interfaces to allow zone-relative pointers in interfaces.

The ZONE type constructor takes a size argument that specifies the amount of address space to be reserved for the zone, and implicitly defines the amount of storage required by a zone-relative reference. It allows optional specification of zone defaults for attributes not supplied by particular calls to NEW. The primary System zone will have the value of the MDS register as its origin.

Zone-relative references can be automatically widened to otherwise similar unzoned references (by adding the origin); narrowing is subject to a run-time range check. Every zone will include an address for NIL, so NIL may always be narrowed or widened without error. Zones represent disjoint storage, so references relative to different zones never conform. (Note that since the zone is part of the *type* of a zone-relative reference, it may be necessary to create multiple instances of a module in order to represent a family of logically equivalent, but physically distinct, data structures that are each internally tied together by zone-relative references; an example is the Mesa symbol-table machinery.)

Referent space = {collectible, heap, internal}: If no referent space is specified, the default is collectible. If the reference is to contain a tag identifying the referent space, it is specified by GENERAL. Internal references are stored as <reference, offset> pairs. Collectible references can be widened to internal references with offset zero; similarly, heap, collectible, and internal references can be widened to general references. The inverse narrowings are subject to run-time range checks. Heap references and collectible references are not inter-convertible.

The system maintains the illusion that all references to collectible storage are counted, while employing a number of tricks to avoid doing very much counting. References to heap storage are not counted, since this storage is freed explicitly. Internal references are useful for referencing such things as array elements and fields of records—an important special case is program variables, which are fields of frame records. The smallest unit of deallocation is an entire allocated object, so an internal reference is counted as a reference to the entire object.

Dynamic type encoding mechanism = {unencoded, prefix, quantum, hash}: This is an aspect of the run-time representation of references that is primarily of concern to the storage management and reclamation subsystem; the tag is carried with each reference value, but becomes visible to the programmer only if a narrowing operation fails due to incompatible mechanisms in the value and the target type.

When collectible storage is allocated by NEW, its type will generally be recorded by either the prefix word or the quantum map mechanism; in the former, an extra word containing the type is allocated just before the object, in the latter, the object is allocated with an address in a range containing only objects of that type, so the high-order part of the address implies the type. Statically typed heap references will not be required to conform to this scheme, and may have the unencoded attribute. However, if such a value is widened to a general or universal reference this is unacceptable; we avoid the need to allocate storage for the type in all such references by entering the <address, type> pair in a special hash table and setting the tag in the reference to hash.

The hash mechanism is the most general, and also the most expensive. Values with the other three tags can always be widened to hash, but the inverse narrowings are subject to run-time checks that the specified mechanism applies. If no mechanism is specified in the call to NEW or the zone declaration, the default will be prefix for large objects, and quantum for small ones.

Static typing = {specific, any}: Cedar Mesa does not allow "universal" variables whose type is not known statically. However, *universal references* may be specified by REFERENCE TO ANY, which indicates that the type will be derivable from the reference value. The only operations on universal references are allocation and deallocation, discrimination on the current type (in a SELECT statement with type names as the adjectives), assignment (including parameter passing), and test for equality. Note that undiscriminated universal references may *not* be dereferenced. References of any specific

type may be widened to universal references; narrowing is subject to a run-time type identity check.

Abbreviations: "POINTER" is accepted as an abbreviation for "System HEAP REFERENCE". "LONG POINTER" is accepted as equivalent to "HEAP REFERENCE". Unqualified uses of REFERENCE denote unzoned collectible references.

Exported Type References

As detailed in the referenced Objects/Exported Types Proposal, references to exported types provide a type-safe mechanism for supporting an object-oriented programming style generally similar to that of Humus. A reference to an exported type may be either *concrete* or *abstract*. A concrete reference type is obtained from a particular module instance that exports the type; the exporting module instance is considered to be part of the type, which makes concrete reference types obtained from separate implementations distinct. An abstract reference type is obtained from a definitions module by specifying ANY; every abstract reference value implicitly carries a reference to an implementing module instance, as well as an explicit reference to data. Concrete references can always be widened to abstract references with the same type name declared in the same definitions module; narrowing requires a run-time test for identity of implementing module instances.

The "Reference to" operator, @

Despite the relatively high frequency with which @ is used in Mesa programs for such useful purposes as passing arguments by reference, we had expected that it would have to be banned from the safe subset—or at least greatly restricted. Somewhat surprisingly, it now appears that our partitioning of the space of reference types has made @ completely safe.

@ always produces a reference, the particular variant depending on the nature of the LeftSide value to which it is applied:

> For any component of a heap variable (i.e., if the last dereferencing operation on the access path was applied to a heap reference), @ produces a heap reference.
>
> In all other cases (including application to an explicit program variable), @ produces an internal reference.

@ is still functionally adequate for passing arguments by reference, but will be substantially more expensive in Cedar Mesa than standard Mesa when applied to explicit program variables. Hence, the addition of VAR parameters to the language is still under consideration.

Reference Assignment

We place no restrictions on assignments of reference values, other than identity of type referred to, plus the various run-time checks on narrowing that have already been mentioned. For safety in the presence of multiple processes, reference assignment (including any required reference-counting transactions) will be made into an atomic (non-interruptible) operation.

NEW

The present Mesa storage management facilities deal with "free pointers" (POINTER TO UNSPECIFIED), and hence cannot be used within the safe subset. To provide a safe mechanism for storage allocation *and initialization* we have extended the NEW operation to take a reference type as its argument. NEW's value is a reference of the argument type that points to a properly-initialized object that does not overlap with any other currently allocated object.

The new object will be in heap or collectible storage, depending on the type of the argument or the zone default; in the absence of either, the default is collectible. Of course, the argument type may not be a universal reference, since that would not provide the information needed to allocate an instance. If the argument type is zone-relative, the allocator will place the object in the corresponding zone.

FREE

The various *free* operations provided by the present Mesa storage management facilities (FreeHeapNode, FreeHeapString, etc.) have been replaced by the FREE statement, which provides the only means of deallocating heap storage. It may be applied only to heap references, and is allowed only in trusted regions. It is always unsafe to free storage if references to it have been passed to safe regions, since there is no way to ensure that copies of references will not be kept. Storage should only be freed if all code that has access to it is trusted.

Although this is a strong restriction, and makes the use of heap references much less attractive, we have not found any acceptable alternative restrictions that would make it possible to explicitly and safely free storage. This is the cost of allowing heap reference assignments.

Persistent Closures

Mesa currently allows local procedures to be assigned to global procedure variables. However, if such procedures reference non-local variables, the result can be arbitrary, since the frames referenced via the procedure's static links may have been deallocated on return from its enclosing procedures. Rather than outlaw the technique, we will make it safe in Cedar Mesa. Procedure values and static links will be treated as references, and frames will (conceptually) be reference counted like other collectible objects. (Of course, we intend an efficient implementation for the overwhelmingly common case where frames obey a stack discipline, and can be freed on procedure return, but this should be transparent to the programmer.)

Run-time Type System

Much of the foregoing has implied the necessity of determining types at run time. Actually, there are two distinct requirements that will be met somewhat differently: sometimes it is sufficient to compare two types for identity—we expect that 16-bit unique identifiers will serve quite nicely; at other times, particularly during garbage collection, it is necessary to locate all the ^C fields in an object and to determine the type of object that each address refers to—for this purpose a (somewhat compressed) version of the compiler's symbol tables will need to be indexed by the unique type identifiers. (Note that the types of frames are a crucial part of this scheme.)

Other Safety Changes

Bounds and NIL Checking

The Mesa compiler has options for generating bounds checking for all assignments to subrange variables and all indexing operations, and NIL checking for all dereferencing operations. Whether or not they are enabled, the compiler will generate checks for all indexing and dereferencing operations in safe regions that are used to access ^C types or to store values.

Most NIL checks can be performed at low overhead by the paging hardware. The entire page (perhaps even several pages) to which NIL's address points will be reserved for this purpose. The

representation of NIL will uniformly be a zero of appropriate length; both widening and narrowing operations must check for NIL and convert it appropriately.

Variant Records

COMPUTED and OVERLAID variant record types provide another way to achieve the effect of a LOOPHOLE. None of the variants in any COMPUTED or OVERLAID variant record type used in a safe region is allowed to be AC.

In present Mesa, even ordinary variant record types with tag fields provide opportunities for type confusion if the tag is changed within the scope of a discriminating selection. Although Mesa prohibits direct assignment to the tag field, there are a number of underhand ways of causing the tag to change, such as calling a procedure that changes a global variable or accessing the record through a different name (aliasing). Even worse, such actions may be triggered by interrupts even if the selection appears perfectly innocuous. Changing the language to prohibit all such possibilities would have required a rather more thorough redesign than we were prepared to attempt.

We distinguish two different sorts of unbound variant record types: those where the variant is *immutable*, i.e., fixed when the object is allocated, and those where the variant is *mutable*, i.e., changable by assignment to the value. If T is a variant record type, unqualified use of T denotes the former case, and ANY T the latter. Of course, T values (like bound T values) may be freely assigned to ANY T variables; assignment in the reverse direction will require a run-time check that the current tags are equal. However, REFERENCE TO T values (and REFERENCE TO bound T values) may *not* be assigned to REFERENCE TO ANY T variables (since this would allow assignments that attempted to change the tag); assignment in the reverse direction is also disallowed (since it would make a mutable variant appear immutable).

In the safe subset, discriminating selection will only be allowed on (references to or values of) immutable variant record types. Furthermore, the semantics of the OpenItem will be changed so that the expression is evaluated only once. Thus there will be no possibility that the tag can be changed between the discrimination and the use of the variant record. For convenient discrimination of mutable values we have introduced a new kind of SelectHead of the form identifier := Expression, which causes the identifier to denote an immutable variant copy of the value of the (potentially) mutable variant expression throughout the scope of the selection.

Assignment of a mutable variant record value will be made into an atomic operation whenever any of the variants are AC, so that an inconsistency between the tag and the variant cannot occur as the result of an unfortunate interrupt.

Since discriminating selections cannot be made on mutable records, there is no way in the safe subset to apply the operator @ to a variant field of a mutable record.

Anonymous OPEN

An OpenItem without the optional identifier is allowed only if none of the identifiers it makes accessible is already known in that scope. This prevents a common source of "hole in scope" errors. In general, the use of anonymous OPENs is discouraged.

Unsafe Features

Although some of them have safe uses, the following Mesa language features are restricted to the unsafe subset:

LOOPHOLE

Arithmetic to produce pointers

DESCRIPTOR

FREE

MACHINE CODE

RELATIVE POINTERs and BASEs will be replaced, even in the unsafe language, by zone-relative references and zones.

Many procedures provided by the (unsafe) Mesa run-time system have unsafe interfaces, most notably UnNew, UnNewConfig, SelfDestruct, and similar procedures.

Remaining Issues

IO with Type Preservation

Certain Cedar packages, most notably the data base package, will wish to treat secondary storage as strongly-typed, so that IO cannot introduce type confusion. This raises a number of important research issues (e.g., what to do about disk files when the definitions of some of the types stored in them change), and we do not presently see how to solve this problem within the safe language. It seems likely that low-level IO routines will have some trusted regions in any case.

Programmer-specified Relocation

In the presence of multiple processes, "finger"-based schemes for storage allocation and relocation can go badly awry if storage happens to be relocated by one (trusted) process while another process is part way through the access chain. We encourage the use of collectible storage, instead.

Summary of Syntactic Changes

Block ::= BEGIN *SafetyClause* OpenClause ...
SafetyClause ::= empty / SAFE / TRUSTED

Access ::= empty | *SafetyClause* PUBLIC | *SafetyClause* PRIVATE

CatchItem ::= *SafetyClause* Catch | *SafetyClause* ANY => Statement

LeftSide ::= ... | LOOPHOLE [Expression *OptTypeSpecification*] |
PUN [Expression *OptTypeSpecification*] |
NARROW [Expression OptTypeSpecification OptCatchSeries]
OptTypeSpecification ::= empty / , TypeSpecification
OptCatchSeries ::= empty / ! CatchSeries

TypeConstructor ::= ... | *ReferenceTC* | *ZoneTC* | ...

ReferenceTC ::= Ordered ZoneOption ReferencePrefix REFERENCE TO ReferenceTail |
Ordered POINTER TO PointerTail
ZoneOption ::= empty / Variable
ReferencePrefix ::= Referent Mechanism
Referent ::= empty / GENERAL / COLLECTIBLE / HEAP / INTERNAL
Mechanism ::= empty / UNENCODED / PREFIX / QUANTUM / HASH
ReferenceTail ::= PointerTail | FRAME [identifier]
PointerTail ::= ReadOnlyOption TypeSpecification | *ReadOnlyOption ANY*

ZoneTC ::= ReferencePrefix ZONE [Expression]

Adjective ::= identifier | *ANY*

NewExpr ::= NEW Variable OptCatchPhrase | *NEW TypeSpecification OptCatchPhrase*

Statement ::= ... | *FreeStmt*

FreeStmt ::= FREE Expression

SelectStmt ::= *SelectHead* FROM
StmtChoiceSeries
ENDCASE FinalStmtChoice
SelectHead ::= SELECT LeftItem |
WITH *OpenItemOrCopy* SELECT TagItem FROM
OpenItemOrCopy ::= OpenItem | *identifier := Expression*

Inter-Office Memorandum

DRAFT - DRAFT - DRAFT - DRAFT

To	File	Date	January 29, 1980
From	David K. Gifford	Location	Palo Alto
Subject	References and Indexes	Organization	CSL

XEROX

Filed on: [ivy]<Gifford>fs>references.bravo

1. Introduction

A framework for object location in a decentralized computer system is presented. Here are some terms I will use repeatedly:

Unique Name A sixty-four bit unique identifier that is generated with a decentralized algorithm. No information is contained in a name.

```
System.UniqueID: TYPE = PRIVATE ...
```

Address The packet transport mechanism can only deliver packets with an address.

```
System.NetworkAddress: TYPE = PRIVATE ...
```

Object An entity (e.g. file, volume) that may reside at one or more machines in the network. Every object has a unique name.

```
File.ID: TYPE = System.UniqueID;
Volume.ID: TYPE = System.UniqueID;
```

2. Indexes

Indexes are not normally used by clients, but I will describe them first as references depend on them.

An index is a function that maps an arbitrary key into an arbitrary value. There are many indexes, and to lookup up a key in an index the index's name must be known.

```
Index.ID: TYPE = PRIVATE CARDINAL;          -- is 16 bits enough?
```

If Lookup does not find key in index then valueSize is zero.

```
Index.Lookup: PROCEDURE [index: Index.ID, key, value: DESCRIPTOR FOR ARRAY OF WORD,
    valueSize: CARDINAL];
```

For example, if a client of the index mechanism wanted to look up unique identifiers, he would have to remember the following structure.

```
Index.Reference: TYPE = RECORD [
```

```
    index: Index.ID,
    id: System.UniqueID
    ];
```

A lookup is accomplished by first broadcasting on the local network on the index socket. Two types of machines can respond to this initial request: the machine that originally placed value in index may respond, even though it does not have a complete copy of index. An index server can also respond, either with value, or the address of a server that services index. Every network segment has an index index server that provides a fixed-point for this structure.

There can be more than one copy of an index, and no consistency checking is done on insertions. Inserting a zero length value is equivalent to deletion. The index is not required to remember the insertion after expiration.

```
Index.Insert: PROCEDURE [index: Index.ID, expiration: System.GreenwichMeanTime, key, value:
    DESCRIPTOR FOR ARRAY OF WORD];
```

Insert sends the key, value, expiration triple to an index server that services index. Once the server accepts the insertion, it guarantees that the insertion will be made in all of the other servers that service index.

3. References

3.1 Client Interface

A new idea I introduce is that a *reference* is required to locate an object. Previous architectures have assumed that possession of a unique identifier would be sufficient to locate an object; I take the simpler view that a client must present a reference.

A reference is a sealed record created and managed by the Kernel, but stored by the client. There are different types of references for files, volumes, and so on. A reference allows the Kernel to deduce an object's address. A reference can have additional properties such as protection information, and the duration of its useful life. Additional properties are omitted from this discussion.

I present a simple file reference, with the expectation that the reader will be able to generalize the idea to other objects.

```
File.Reference: TYPE = PRIVATE RECORD [
    file: Index.Reference,
    volume: Index.Reference,
    permissions: File.Permissions
    ];

File.ReferencePtr: TYPE = POINTER TO File.Reference;
```

Before a file can be accessed, its reference must be opened. After the client is done with the file, its reference must be closed.

```
File.Open: PROCEDURE [rp: File.ReferencePtr];

File.Close: PROCEDURE [rp: File.ReferencePtr];
```

A client can seal, unseal, and copy file references:

```
File.SealReference: PROCEDURE [file: File.Capability, volume: Volume.Reference, rp:
    File.ReferencePtr];

File.UnsealReference: PROCEDURE [rp: File.ReferencePtr] RETURNS [file: File.Capability];
```

```
File.CopyReference: PROCEDURE [rp: File.ReferencePtr, copy: File.ReferencePtr];
```

References for other objects, such as volumes, can be manipulated in a similar manner.

```
Volume.Reference: TYPE = PRIVATE RECORD [
    volume: Index.Reference,
    permissions: Volume.Permissions
    ];
```

3.2 How a file reference works

When a file reference is opened, *Lookup*[**volume**] gives us the address of the volume where the file is supposed to be. We check to see if the file is on the volume. If so, we have found the file. Otherwise, if *Lookup*[*Lookup*[**file**]] is not null it is the address of the volume that should contain the file. If the file is located there, we update **volume** to be *Lookup*[**file**].

A file can be moved from one volume to another as long as the movement will not void any guarantees that have been made. When a file is moved from one volume to another, we leave a pointer on the original volume (in the form of a "truncated" weak representative), and also insert the triple [File.ID, *new* Volume.ID, **time**] in the index **file.index**. The expiration time **time** is chosen as a function of the guaranteed properties of the file's references.

CSL Notebook Entry

To	CSL Cedar Interest Group	Date	February 15, 1980
From	Cedar Language Features Design Committee	Location	PARC/CSL, Palo Alto
Subject	Cedar Mesa—Version 6C2	Files	[Ivy]<CedarDocs>Lang>6C2> *.bravo, Design.press

XEROX

Archive ID: yyCSL-xxx

Attributes: technical, Cedar, Mesa, Programming research

References: [Ivy]<CedarDocs>Lang>Proposal.press (September 1, 1979)
Cedar Mesa Proposal---5C2 (Drafts November 1979-January 1980)
Mesa Language Manual, Version 5.0, CSL-79-3, April 1979

Abstract: This document summarizes our current design for the version of Mesa to be provided by Cedar. Its motivation and required properties are described. Specific changes to Mesa fall into three major areas: safe language, delayed binding, and exported types. The specific language design described here is called Mesa 6C2; although it provides the basis for the initial implementation, it may not correspond exactly to any implemented version. Mesa 5C1 refers to the proposal dated September 1, 1979; Mesa 5C2 refers to the series of draft proposals from October 1979 through January 1980; Mesa 6C3 is a catchall term for good ideas whose implementation (or detailed design) has been deferred, although we expect to incorporate them in some future version of Cedar Mesa.

TABLE OF CONTENTS

Introduction	2
Language Changes	
Safe Language	8
Delayed Binding	19
Exported Types and Objects	22
Garbage Collection Invariant	30
Notes for Implementation	33
Grammar	37

INTRODUCTION

The design presented here evolved through considerable discussion, taking into account reactions to the 5C1 and 5C2 references cited above. 5C1 was circulated to CSL and the Cedar Interest group. The Cedar External Design Review in September also provided specific feedback on 5C1. This design (6C2) provides the basis for the initial implementation and use of Cedar Mesa.

Cedar Mesa will be very closely related to *current Mesa* (the 5Cx proposals were based on Version 5.0, the 6Cx designs, on Version 6.0), as implemented and supported by SDD. Probably the largest single extension is the provision of garbage-collected dynamic storage (and a concomitant reduction in the role of heap storage). It is intended that most code will be written in the *safe subset*, which imposes a number of restrictions not present in current Mesa to ensure the safe operation of the garbage collector, and introduces some new (safe) features to make these restrictions tolerable. The full (unsafe) language is generally "upward compatible" with current Mesa.

We have generated many further ideas for the improvement of Cedar Mesa, and received large numbers of suggestions. However, we believe that the present design addresses the most pressing issues in (at least) a satisfactory manner, and that it is more important to procede with the timely implementation of this design than to further refine it. The Cedar implementors will expend considerable effort coping with the major limitations of current Mesa until Cedar Mesa is available. They also need a stable definition of Cedar Mesa, so they can plan their implementations accordingly. This document is intended to be that stable definition—although some changes may still be made if compelling reasons are discovered, and some features may be deferred if problems arise.

We are still collecting suggestions for improvement; we believe that the availability of a stable implementation of 6C2 will speed the incorporation of further changes. We intend to commence a careful design of Cedar Mesa 6C3 in fall 1980, and to issue further proposals at that time.

SAFE LANGUAGE

Motivation

A desirable property of a high-level language is *implementation independence*, which means that the effects of each program are explicable in terms of the *language* (rather than its implementation)—even if the program is erroneous. Mesa comes rather close to meeting this goal (as evidenced by the fact that most Mesa programs can be debugged "at the Mesa level," without ever worrying about the format of stack frames or the details of storage management), but it does contain some "unsafe" features whose use can lead to messy implementation dependencies.

It would be desirable to reduce implementation dependencies in Cedar on general grounds. However, the decision to include facilities for automatic storage deallocation ("garbage collection") within Cedar Mesa makes a shift in this direction imperative. The collector can cause storage to be deallocated (permitting its subsequent reallocation and re-use) at times that are completely unpredictable from examination of the source program. A single programming error that smashes a reference used by the collector can effectively bring the whole system down in a rubble of bits, from which it is difficult to reconstruct any evidence of the cause of the crash.

It has been one of the major goals of the Cedar Language Features Design Committee to design a subset of Mesa for which it would be safe to do automatic storage deallocation. To first

approximation, the "safe subset" is that part of the language where even incorrect programs cannot interfere with the reliable operation of the collector.

We intend that the vast majority of programs will be written primarily in the safe subset. However, we recognize that it is not presently feasible to provide acceptably efficient substitutes for all uses of Mesa's unsafe features. Instead, we provide textual means for indicating that some regions of a program are *unchecked.* This will inhibit compiler enforcement of the safety restrictions and simultaneously indicate that the programmer has assumed the additional responsibility of ensuring that these regions of the program cannot violate the integrity of the system.

Invulnerability, Safety, and Checking

It is an obviously desirable property of a programming system that no user programming error can reduce its world to a rubble of bits. We will call this property *invulnerability.* In general, it can be ensured only by maintaining the integrity of certain data structures known to the run-time system (e.g., dispatch vectors, core maps). Collectively, the properties that must be maintained to ensure invulnerability will be called the safety *invariants;* each portion of the system will be responsible for ensuring that they are not destroyed, and will be free to assume that the rest of the system does likewise.

Invulnerability is not a local property. If any part of the system fails to maintain the invariants, the entire system (including regions that are themselves correct) is potentially vulnerable. We use the term *safety* for the property that the invariants cannot be invalidated locally (even by incorrect programs). *Checked* program regions are portions of a system where safety is guaranteed by language-enforced restrictions. Even *unchecked* regions are supposed to preserve the invariants, but the guarantee must be provided by the programmer, rather than by the language and its implementation. If a system consists entirely of checked regions (and the invariants hold initially), then by induction the program is invulnerable. However, an error in an unchecked region could make even the checked regions vulnerable. Thus the checked/unchecked boundary limits responsibility, but not vulnerability.

Unchecked regions may use unsafe language features, and errors may cause violation of system invariants. These regions are sometimes called "trusted regions," because confidence in the correct operation of the system must be based on trust that they are free of such errors. By contrast, our confidence that errors in checked regions will not cause system crashes is based on the knowledge that the safety restrictions have been automatically enforced.

Type Confusion

Mesa is a strongly typed language, which means that the types of variables are declared, and that the language imposes restrictions to keep values of one type from being accidentally interpreted as values of another. Because knowledge of the type structure of memory is so essential to the collector (it must locate and follow references in order to determine storage usage), it is particularly vulnerable to any operations that cause things in memory to be interpreted as having other than their declared types. Thus, much of the effort in designing the safe subset has gone into identifying all the features in the present Mesa language that allow type-checking to be circumvented (accidentally or deliberately) and designing safe replacements for important uses of those features.

LOOPHOLE, of course, is the most obvious means of defeating type-checking. It causes a safety problem if it allows mis-typing of some piece of memory (i.e., if the target type contains an address, such as a reference or a procedure value); other uses will introduce implementation dependencies that do not threaten safety. We have formalized the distinction between these two sorts of breach of the type system, retaining LOOPHOLE in the unsafe language for arbitrary reinterpretation of the type

of a value, and introducing the safe language type transfer function PUN, which is semantically equivalent to LOOPHOLE, but is restricted to prevent misuse of address-containing types.

Narrowing

This design introduces a number of new type distinctions into Mesa, frequently leading to a number of separate, but closely related types. It is often desirable to *coerce* a value of one of these types into a value of a related type. If it can be statically guaranteed that no information can be lost by such a conversion, we call it a *widening*. Widening will be performed automatically whenever demanded by context—this is analogous to lengthening an INTEGER to a LONG INTEGER. In general, conversion in the other direction will require a run-time check to ensure that information is not being lost. For each such *narrowing* described below we indicate the nature of the necessary check; if the check fails, the signal RangeError will be raised. To make the possibility of such failure explicit in the program text, the NARROW type converter may be applied when a narrowing is to be performed; it will also be applicable to present Mesa types (e.g., LONG INTEGER).

Implicit narrowing will be supplied automatically if the target type is uniquely determined by context. An explicit narrowing may require an automatic widening of its argument (e.g., from a concrete to a generic type) before the narrowing; however, the combination of widening followed by narrowing will not be supplied automatically.

References and Garbage Collection

Although Mesa pointers are nominally typed, they provide a rich source of opportunities for type confusion, including—but not limited to—the classical "dangling pointer" problem, where a pointer persists after the storage it refers to has been deallocated. We are introducing a new class of *reference* types that refer to garbage collected (automatically deallocated) storage.

In the unsafe language, the UNCOUNTED attribute may be applied to types that conceptually contain references to produce corresponding types that are not counted (i.e., traced by the garbage collector). POINTER TO will be a synonym for UNCOUNTED REF. Pointers may not be used in the safe subset, and cannot refer to objects containing references.

VAR Parameters

One common use of pointers in current Mesa is to pass parameters to procedures "by reference," rather than "by value"—either to allow the results of assignments to the formal parameters to be visible in the calling context, or to eliminate the overhead associated with copying large arguments. Mesa 6C2 provides a special parameter attribute, VAR, for this purpose. In checked regions, the "address of" operator, @, is only allowed in the construction of arguments for VAR parameters. This restriction greatly reduces the cost of ensuring the safety of the storage reclamation mechanism.

DELAYED BINDING

Motivation

A desirable property of a high-level language is that it allow a wide range of *binding times*: that is, it should allow the programmer maximal control over when the attributes of a particular variable are determined, with different choices not requiring changes at each occurrence of the variable's identifier. Examples of attributes include type, storage allocation method, implementation (of abstract objects), and actual value; examples of binding times include program-writing time, compilation, program initialization, block entry, and dynamic assignment. Generally speaking, deferring the binding of an attribute leads to greater generality in the program at the cost of decreased static checkability and (often) runtime efficiency.

Experience with languages like Lisp and Smalltalk, which provide only very late binding, suggests that even though late binding may actually be required only in few situations and have significant runtime cost, there are certain kinds of programs that are extremely clumsy to write if late binding of type or implementation is not available. In particular, programming tools (debuggers, performance monitors) and knowledge representation systems seem to be of this nature. Thus, in addition to wanting more binding time flexibility in Cedar Mesa on general grounds, we believe such flexibility will greatly simplify the writing of an important class of programs.

Dynamic typing

Mesa provides very limited variability in the binding time of an object's type. Variant records allow a deferred choice between specific enumerated alternatives, and string and array descriptors allow deferring the choice of an object's length. Otherwise, all types must be fixed at compile time. This requirement makes it impossible to avoid LOOPHOLEs and *ad hoc* type tagging schemes when writing schedulers, searchers, printers, etc. that must operate on objects of unpredictable type. Our solution to this problem requires two new mechanisms: a runtime representation for types, and a way to associate a type with an object at runtime that is guaranteed consistent with the type system established at compile time. Note that we explicitly embrace the traditional view that an object's type is inherent in the object itself (rather than the Russell view that its type is a syntactic property of the variable through which it is referenced).

For runtime types to be useful, programs must be able to perform at least a limited set of operations on types (e.g., equality comparison). We intend to provide a logically complete set of such operations in the future. We view the operation that forms the type REF T from the type T as simply another operator in the language, like +. In Mesa 6C2, we have kept the current limitation that the arguments to such operators must be compile-time constants, but it is a semantic limitation, not a syntactic one.

Lists

We have added a new type constructor, LIST OF, to declare singly-linked lists whose elements are all of the same type. The record constructor notation is extended to construct list values in contexts that require them.

EXPORTED TYPES AND OBJECTS

Motivation

Current Mesa systems are constructed by binding together collections of separately written and compiled modules that communicate through explicit and precisely specified interfaces. The language has been designed so that specifications of operations, which appear in interfaces, are decoupled from their implementations, which are hidden within program modules. Specifications of types do not currently have similar properties; all information about types used in interfaces must appear within the interface definitions themselves and must be fixed before any client code can be compiled. This shortcoming violates the principle of information hiding and also creates awkward compilation dependencies.

Support of the so-called "object-oriented" style of programming, as developed in such languages as Simula and (especially) Smalltalk, is an important goal of Cedar. Attempts to use Mesa for such programming particularly suffer from the inability to hide representational detail. In addition, Mesa does not distinguish between abstract and concrete types. The result is that dealing with an abstract class of objects with multiple and coexisting concrete implementations requires considerable circumlocution; the code is awkward and error-prone.

Programmers avoid some of the dependencies and awkwardness by relying upon extralinguistic (and thus uncheckable) conventions and then using various LOOPHOLEs to circumvent normal checking. This approach is obviously undesirable on general principles; it is unacceptable if we want most programming to be done within the safe langauge.

Opaque Types

The current technique for hiding the underlying representation of something is to declare its type as a place holder with no useful structure and then to allow implementors to use LOOPHOLEs of some sort to map the values into the actual underlying type. The purpose of introducing opaque types is to institutionalize this well-understood use of LOOPHOLE.

In Mesa 6C2, types may be exported from implementors into interfaces, just as procedures are. An interface may contain a declaration of an *opaque type*, the structure of which is largely unspecified and thus invisible to importers of that interface. Every implementation of the interface must supply a fully specified *concrete type* for each opaque type in the interface, as well as a set of procedures implementing operations upon that type. (All modules in a particular implementation that use the type concretely must export the *same* concrete type.) We do not require that different implementations supply the same concrete type. Mesa's rules for type checking have been modified slightly to prevent the unintentional mixing of different concrete types.

Opaque types allow the implementation of a type appearing in any particular interface to be fixed at the time of binding. They permit the use of different concrete types for a single abstract type (as long as these concrete types are kept well separated) and have the practical benefit of eliminating many compilation dependencies.

Generic Types

Opaque types do not handle the situation in which multiple implementations of an abstract type coexist within a system and clients wish to use the various implementations interchangeably. This case can, in principle, be handled by merging all the concrete types into a variant record type and by explicitly constructing suitable interface (operation) records. The representation of an object is then a pair, with one component indicating a particular variant and the other identifying an interface record. This approach requires either LOOPHOLEs or useless discriminations, leads to awkward notation, and generally suffers from the problems noted above.

Our design extends the opaque type mechanism to allow the intermixing of different implementations. It thus accommodates a more general style of object-oriented programming. For each opaque interface type, we introduce a *generic type*, the values of which can have any of the concrete types supplied for the opaque type. Such a value is represented by a value of the concrete type plus an interface providing its operations.

Notational Conventions

The notation used for object-oriented programming in Mesa depends upon the degree of generality anticipated by the programmer. If only one implementation is anticipated, he uses standard functional notation, e.g., Interface.Op[object, args]. If multiple implementations are anticipated, an object-oriented notation, e.g., object.Op[args] is more appropriate. We introduce conventions that allow these notations to be used interchangeably, given suitable declarations. This allows the programmer to adopt a uniform style and to delay another level of decisions about representation.

Subclasses

Sometimes several classes of objects share a number of abstract operations; these can all be viewed as subclasses of some other class for which just the common operations are defined. Algorithms applicable to the superclass automatically extend to all the subclasses. Part of our design is a means for defining an interface as an extension of other interfaces. The effects of a subclassing can be achieved if one of the interfaces being extended includes an opaque type.

LANGUAGE CHANGES

The changes proposed here are for Mesa 6C2. Omission of a feature does not imply it will not be included in some future extension.

SAFE LANGUAGE

General Principles

System Invariants

Basically, the invariants that must be preserved are

the integrity of the code and the various data structures maintained by the present run-time system for storage management, process scheduling, control links, etc.

the integrity of the new data structures introduced for garbage collection (reference counts, transaction queues, symbol table information, etc.).

the accessibility of all active *reference-containing (RC)* storage to the collector by tracing links, starting from a known set of locations.

the type integrity of all RC storage.

Checked and Unchecked Regions

Each block of a program will belong to a *checked* or *unchecked* region, with checked being the default. An unchecked region may contain embedded checked regions, but not conversely. Our goal is to have checked regions predominate, and to make it easy for the reader to locate all unchecked regions.

We have added the key words CHECKED and UNCHECKED to the language, to precede the BEGIN of a block. Unless a module body's block begins with UNCHECKED, it is automatically checked. Unless a nested block in an unchecked block begins with CHECKED, it is automatically unchecked. UNCHECKED may not appear within a checked block.

Within a checked block, all the restrictions imposed on the safe subset will be enforced by the compiler (possibly generating some run-time checks for such things as array bounds). In unchecked regions, the full language may be used, but the programmer is responsible for ensuring that he both understands and preserves all the system invariants. (The new invariant required for safe garbage collection is described in a separate section.) Even in unchecked regions it is likely that the bulk of most programs can be in the safe subset, and features outside the subset should be used only for good reasons and with due caution, since an error in an unchecked region can bring down all of Cedar.

Unchecked Regions Providing Safe Interfaces

Although a checked module is not allowed to contain any unchecked regions, it may *import* interfaces to unchecked modules. We distinguish between unchecked procedure bodies and unsafe procedure calls. A procedure that uses unsafe features to accomplish some safe operation (i.e., one that will not violate the safety invariants, even in the presence of errors in the calling program) has a

safe interface, and such procedures may be called from within checked regions. Of course, the compiler cannot guarantee that an unchecked procedure body actually provides a safe interface; this is the responsibility of the programmer who declares the interface to be safe.

The key words SAFE and UNSAFE are new access attributes. Preceding a procedure, port, process, or signal type constructor, SAFE indicates an interface that may be used within checked regions. Unsafe interfaces are private to unchecked regions, and invocation from checked regions is prohibited. Interfaces declared in checked regions are automatically safe, while those declared in unchecked regions are unsafe unless declared SAFE.

Signals are like procedures: checked regions are not allowed to raise unsafe signals, and unchecked regions may only catch safe signals with safe catch phrases, i.e., those whose statement is a block that is preceded by SAFE. Note that this SAFE applies to the *interface*. If the block is to be a checked region, it must be preceded by CHECKED (which implies SAFE). An UNWIND, however, may be caught anywhere, because the region that raised it is being terminated.

References and Other Address-Containing Types

AC Types

We need to impose restrictions on the creation and use of values of *address-containing (AC)* types, which are either *reference-containing (RC)* or *pointer-containing (PC)*.

References (addresses) are AC.

String, procedure, frame, descriptor, signal, port, and list types implicitly contain addresses, and are therefore AC.

Counted AC types (the default) are RC, and are the primary concern of the collector.

UNCOUNTED RC types (including pointer types) are PC.

Arrays whose component type is RC (PC) are RC (PC).

Records with any RC (PC) fields are RC (PC).

Whenever a RC value is created, all its address fields must be validly initialized (possibly to NIL) to avoid assigning through, tracing, or deallocating, non-existent structures. In Cedar Mesa, all address fields of RC types that are not explicitly initialized will be implicitly initialized to NIL. (In Mesa 6, type definitions will be allowed to include specifications of instance initializations; this will also override the implicit initialization to NIL.)

To free the garbage collector from concern for PC values, pointers may not refer to RC values; i.e., if T is RC, then POINTER TO T is not a valid type. Conversely, PC types may not be declared or used in checked regions.

We must also guard against both "forgery" of AC values and the preservation of RC values in locations unknown to the collector. The immediate implication is that we must prohibit LOOPHOLE and restrict UNSPECIFIED in checked regions.

As long as there is no mechanism for later converting them back to AC values, there is no *safety* problem in allowing AC values (indeed, any values) to be interpreted as non-AC values, and we provide PUN as the safe replacement for LOOPHOLE.

In checked regions, values of type UNSPECIFIED will not conform to any AC type, although values of AC types may be coerced to UNSPECIFIED. REF ANY provides type-carrying references, which can safely be used to replace many uses of UNSPECIFIED.

Automatic Deallocation and Reference Counting

Cedar Mesa provides three kinds of storage:

> Frame: This is storage that is implicitly allocated at procedure activation or module instantiation to hold variables declared in the corresponding scope. It is also implicitly deallocated—normally on scope exit, although other cases will be discussed below.
>
> Heap: This is storage that is explicitly allocated by NEW, and deallocated by (unsafe) FREE statements. Heap storage is referenced by pointers, which may declared and used only in unchecked regions.
>
> Collectible: This is storage that is explicitly allocated by NEW, and is implicitly deallocated after there are no more accessible references to it. FREE applied to a collectible reference will cause it (and its RC fields) to be "NILed out," but the object will only be freed when no other references to it remain.

For all three types of storage, the smallest unit of deallocation is an entire allocated object.

Present Mesa provides the first two kinds of storage. Freeing the programmer from the necessity of explicitly deallocating storage (and the possibility of making mistakes in such deallocation) has been one of the major goals in the design of Cedar Mesa.

Although the programmer should not be concerned with implementation details of collectible storage, some understanding of the intended implementation will make it easier to understand the restrictions imposed on the safe subset.

The basic approach to garbage collection is the same as that advocated by Bobrow and Deutsch in their Sept. 1976 *CACM* article. Responsibility for reclamation is divided between two collectors:

> Full collector: This program traces the entire active data structure and generates a compacted, but otherwise equivalent, copy. It must discard inaccessible circular structures and cope with pathological cases, but it is allowed to be slow, and will only be invoked when the system is otherwise quiescent, so it can be written knowing that the data structure will not be changed by other processes while it is running.
>
> Incremental collector: This program is intended to reclaim most of the inaccessible storage at much more frequent intervals. It need not reclaim all the inaccessible storage (as long as it "usually" reclaims "most" of it), but must be able to work incrementally with low overhead while other processes are active in the system.

The incremental part of the B-D scheme requires the same information as simple reference-counting schemes. Conceptually, each object in storage has an associated count of the number of references currently pointing to it; objects may be deallocated at any time after their reference count reaches zero. However, the B-D scheme gives special treatment to commonly-occurring cases, greatly reducing the cost of the record-keeping. We expect its efficiency to be good enough that collectible storage will replace most present uses of heap storage.

Variants of Reference Types

Three aspects of each reference are part of its type:

Counting option = {counted, uncounted}: The default for AC types is counted; however, in unchecked regions, UNCOUNTED may be applied to any AC type to produce the corresponding PC type; POINTER TO is synonymous with UNCOUNTED MDS RELATIVE REF.

@ produces an uncounted reference, and hence cannot be used in checked regions (except when creating a VAR parameter, as explained below).

The system maintains the illusion that it actually counts each counted reference, while employing a number of tricks to avoid doing very much counting. Uncounted references should either refer to heap storage, which is freed explicitly, or be "redundant," e.g., refer to objects that are guaranteed to be kept in existence by other (counted) references.

Objects containing pointers (uncounted references) may only be declared and used in unchecked regions. Furthermore, pointers may not refer to objects whose type is RC (although counted references may—in unchecked regions—refer to PC objects).

Packing = {absolute, relative}: By default, references are absolute, and contain full machine addresses (28 bits). However, within the scope of a base, many references can be represented with a smaller number of bits (e.g., 16) as an offset from the base's origin.

Relative references can be automatically widened to otherwise equivalent absolute references (by adding the base's origin); narrowing is subject to a run-time range check. Every base's range includes an address for NIL, so NIL may always be narrowed or widened without error.

> Since the base is part of the *type* of a based reference, the same type cannot be used to describe references relative to different bases. It may be necessary to create multiple instances of a module (or to use unchecked code) in order to represent a family of logically equivalent, but physically distinct, data structures that are each internally tied together by based references; an example is the Mesa symbol-table machinery.

Static typing = {specific, any}: Cedar Mesa does not allow "universal" variables whose type is not known statically. However, *universal references* may be specified by REF ANY, which indicates that the value will be a reference whose type is determined dynamically.

The only operations on universal references are allocation and deallocation, extraction of the current value's type, discrimination on the current type (in a SELECT statement with type names as the adjectives), narrowing to a particular reference type (explicit or implicit), assignment (including parameter passing), and test for equality. Note that undiscriminated universal references may *not* be dereferenced.

> A universal reference may, however, be dereferenced as the argument to a PUN.

REF T conforms freely to REF ANY, and references of any specific type may be widened to universal references. However, (uncounted) references into a heap zone without a dynamic type representation may not be so widened. Narrowing a universal reference to a specific reference type is subject to a run-time type identity check. See the section on delayed binding for further details of the use of universal references.

Compatibility: POINTER TO is accepted as equivalent to UNCOUNTED MDS RELATIVE REF. LONG POINTER TO is accepted as equivalent to UNCOUNTED REF. Unqualified uses of REF denote counted unbased references.

Variants of Reference Values

One aspect of the representation and use of references is a property of particular reference values, and not part of their type:

Dynamic type encoding mechanism = {unencoded, prefix, quantum}: This is an aspect of the run-time representation of references that is of concern only to the storage management and reclamation subsystem. From the zone into which a reference points, it is possible to determine its type encoding mechanism.

When collectible storage is allocated by NEW, its type will be recorded by either the prefix word or the quantum map mechanism; in the former, an extra word containing the type is allocated just before the object, in the latter, the object is allocated with an address in a range containing only objects of that type, so the high-order part of the address implies the type. Uncounted (heap)

references will not be required to conform to this scheme, and may have the unencoded attribute. (However, such a value may not be widened to a universal reference, only to POINTER TO UNSPECIFIED.)

The dynamic type encoding mechanism for a zone is specified when it is created. Generally, prefix is preferable for large objects, and quantum for numerous or small ones.

NIL

Each AC type T has a special value that denotes a "non-address," written NIL[T]. When the type is determined by context, this may be abbreviated as NIL.

Bases and Zones

The BASE type constructor includes a subrange, which specifies the amount of address space to be reserved for each base, and implicitly defines the amount of storage required by any reference based on a value of that type.

New base values can be obtained using an extension of the record constructor notation, optionally specifying a containing base and a size (which may not exceed the span of the type). A base's address space may be allocated to "contained" bases and zones.

The system will supply a predefined base, rootBase, which contains the virtual address space available for dynamic storage allocation. RootBase will contain another predefined base, MDS, which will have the value of the MDS register as its origin and a 16-bit range. MDS is intended for the creation of "short" relative references within the current main data space.

MDS is used by the system itself in various ways, e.g., for the allocation of local frames for procedures.

In checked regions, base variables must be initialized, and cannot be otherwise assigned to, i.e., they are constants. In Mesa 6C2, base variables may not be declared local to procedures.

Examples:

```
newBase1: BASE = [size: n, parentBase: rootBase];
newBase2: BASE [0..32768) = [size: m, parentBase: newBase1];
```

References relative to newBase2 require only 15 bits, while those relative to newBase1 are full addresses.

The ZONE type constructor specifies *heap* or *collectible* management (i.e., explicit or automatic deallocation), with the default being collectible. Heap zones are specified by the adjective UNCOUNTED.

New zone values can be obtained using an extension of the record constructor notation, optionally specifying the dynamic type representation for objects created within the zone, a containing base, and an initial size.

The system will supply two predefined zones, systemZone and systemHeapZone. SystemZone will be the default zone for allocation of collectible storage, will be allocated within rootBase, and will use the prefixed type representation. SystemHeapZone will be uncounted, will be allocated within MDSbase, and will not carry dynamic type information.

Examples:

```
newZone1: ZONE = [typeRep: prefixed, base: newBase1, initialSize: j];
newZone2: UNCOUNTED ZONE = [typeRep: none, base: newBase2, initialSize: k];
```

VAR Parameters

@ is frequently used in Mesa programs to pass parameters "by reference," rather than the "by value" default, either because assignments to the formal parameter should affect the argument, or to avoid the overhead of copying the argument. In Mesa 6C2 we raise this usage to the level of a language feature by introducing a VAR attribute for formal parameters.

Operationally, specifying a parameter to be VAR is equivalent to adding REF to its type, and following each use of its name by ↑. Note that this implies that the procedure does not have access to the reference created for the argument, but only to the value referenced; in particular, the procedure cannot either change the reference or copy it into a reference variable. It is this restriction that allows VAR parameters to be implemented safely at a lower cost than unrestricted references.

If the VAR formal parameter type is ARRAY OF T, (i.e., the array bounds are omitted), the parameter will be treated as a descriptor. In such a case, the argument may be an explicit descriptor (in unchecked regions) or the result of applying @ to either an array or sequence value.

@ may be used in checked regions only to create VAR arguments. That is, it may appear only in arguments corresponding to VAR formal parameters.

The "Pointer to" Operator, @

In unchecked regions, @ may be applied to any variable to produce a pointer that refers to it.

> As in current Mesa, @ may be applied to a component of a (record, sequence, or array) value only if it is word-aligned.

It is a safety requirement—to be ensured by the unsafe language programmer—that these pointers not *dangle*:

> A heap object that (has a component that) is the referent of an accessible pointer may not be freed.
>
> A procedure may not return leaving any accessible pointers to its local variables.
>
> If @ is applied to (a component of) a collectible object, a reference to the entire object must remain accessible for at least as long as the pointer does.

Reference Assignment

For safety in the presence of multiple processes, reference assignment (including any required reference-counting transactions) will be made into an atomic (non-interruptible) operation.

NEW

The present Mesa storage management facilities deal with "free pointers" (POINTER TO UNSPECIFIED), and hence cannot be used within the safe subset. To provide a safe mechanism for storage allocation *and initialization* we have extended the NEW operation to apply to any type, with optional tag/length and zone arguments.

NEW's value is a reference that points to a properly-initialized object of the argument type that does not overlap with any other currently allocated object. The new object will be in the specified zone, and the reference will be counted or uncounted, depending on the zone. The default zone is systemZone. The tag argument for variant record types specifies the variant to be allocated. The length argument for sequence types (including TEXT) specifies the number of elements in the allocated sequence.

Examples:

... NEW T ...	--Returns a REF T in systemZone
... NEW T [newZone1] ...	--Returns a REF T in newZone1
... NEW T [newZone2] ...	--Returns a LONG POINTER TO T in newZone2, which can be narrowed to 15 bits
... NEW TEXT[ln] [newZone1] ...	--Returns a REF TEXT of length ln in newZone1
... NEW TEXT[50] [newZone2] ...	--Returns a LONG POINTER TO TEXT of length 50 in newZone2

FREE

The various *free* operations provided by current Mesa's storage management facilities (FreeHeapNode, FreeHeapString, etc.) have been replaced by the FREE statement, which provides the only means of deallocating heap storage. It will usually be applied to pointers, which are allowed only in unchecked regions. (It would be unsafe to free storage if pointers to it had been passed to checked regions, since there is no way to ensure that copies of pointers will not be kept; unchecked regions must be "trusted," anyhow.)

Storage should only be freed if all code that has access to it is trusted. Although this is a strong restriction, and makes the use of pointers much less attractive, we have not found any acceptable alternative restrictions that make it possible to explicitly and safely free storage. This is the cost of allowing pointer assignments.

FREE may also be applied (even in checked regions) to counted references, but in this case serves only as a hint to the allocator that the programmer believes that this is the last accessible reference. FREE's argument and any component RC fields of the object it refers to will be reset to NIL, but the storage will only be freed when no accessible references to it remain. This is a safe and useful way to break circular structures.

Persistent Closures

Mesa currently allows local procedure values to be assigned to global procedure variables and to be returned as the results of procedures. However, if such a local procedure references non-local variables, the result can be arbitrary, since the frames referenced via the its static links may have been deallocated on return from its enclosing procedures. Rather than outlaw the technique, we will make it safe in Cedar Mesa. Procedure values and static links will be treated as references, and frames will (conceptually) be reference counted like other collectible objects.

Other Safety Changes

Bounds and NIL Checking

The Mesa compiler has options for generating bounds checking for all assignments to subrange variables and all indexing operations, and NIL checking for all dereferencing operations. In Cedar Mesa, the compiler will generate checks for all indexing and dereferencing operations in checked regions that are used to access RC types or to store values, whether these options are selected or not.

PUN and NARROW

Although LOOPHOLE is prohibited in checked regions, there is no safety problem in allowing one type to be interpreted as another as long as

> both types occupy the same amount of storage, and
>
> each AC field of the target type conforms freely to the corresponding field in the source type.
>
> An opaque type from an unsafe interface may not be the target type of a PUN, since the concrete type is potentially PC.

If these conditions are met, PUN may be used as a type transfer function, even in checked regions. Like LOOPHOLE, it merely causes a reinterpretation of the type of a value, it does not cause any code to be generated.

NARROW, on the other hand, causes a run-time check that a value known statically to be of some general type (e.g., REF ANY, ANY Foo.Fie), actually contains a value of a specified type. If the test fails, the signal RangeError will be generated. The type need not be specified explicitly if it can be statically inferred from the context; in such cases the explicit NARROW is optional (unless a catch phrase for the signal is to be attached). NARROW applied to a value of an opaque type (see the section on exported types) will first widen its argument to the generic type before narrowing; this allows a programmer to take advantage of the knowledge that two instances of an interface supply the same concrete type (or to confirm that they do).

> Widening is always implicit, and supplied automatically when demanded by context.

Variant Records

COMPUTED and OVERLAID variant record types provide another way to achieve the effect of a LOOPHOLE. COMPUTED and OVERLAID variant record types may be used in checked regions only if each of the variants may be PUNed into each of the others, i.e., if confusion about the variant cannot change the interpretation of the AC fields.

In current Mesa, even ordinary variant record types with tag fields provide opportunities for type confusion if the tag is changed within the scope of a discriminating selection. Although Mesa prohibits direct assignment to the tag field, there are a number of underhand ways of causing the tag to change, such as calling a procedure that changes a global variable or accessing the record through a different name (aliasing). Even worse, such actions may be triggered by interrupts even if the selection appears perfectly innocuous. Changing the language to prohibit all such possibilities would have required a rather more thorough redesign than we were prepared to attempt.

We distinguish two different sorts of unbound variant record types: those where the variant is *immutable*, i.e., fixed when the object is allocated, and those where the variant is *mutable*, i.e., changeable by assignment to the value. If T is a variant record type, unqualified use of T denotes the former case, and ANY T the latter.

Of course, immutable T values (like bound T values) may be freely assigned to ANY T variables; assignment in the reverse direction will require a run-time check that the current tags are equal. However, REF T values (and REF bound T values) may *not* be assigned to REF ANY T variables (since this would allow assignments that attempted to change the tag); assignment in the reverse direction is also disallowed. In the terminology of the Mesa manual, T conforms to ANY T, but does not freely conform.

> The problem with assigning a REF ANY T value to a REF T variable is that if a program in one process discriminates on the latter (which appears to be a safe operation), and then a higher-priority process assigns a new variant to the object referred to by the former, the tag could change between the discrimination and the use.

In checked regions, discriminating selection without copying will be allowed only on (references to or values of) immutable variant record types. Furthermore, the semantics of the OpenItem will be changed in Cedar Mesa so that the expression is evaluated only once. Thus there will be no possibility that the tag can be changed between the discrimination and the use of the variant record. For convenient discrimination of mutable values we have introduced a new kind of SelectHead of the form identifier := Expression, which causes the identifier to denote an immutable variant copy of the value of the (potentially) mutable variant expression throughout the scope of the selection.

Assignment of a mutable variant record value will be made into an atomic operation whenever any of the variants are AC, so that an inconsistency between the tag and the variant cannot occur as the result of an unfortunate interrupt.

> We have removed the syntactic restriction that the variant ("union") type constructor can only appear inside a record constructor. However, we retain the restriction that there can be at most one variant constructor within a record.

Strings, Text, and Sequence Types

Sequences of characters are a frequently-used programming construct. In present Mesa, STRING denotes a pointer to such a sequence (generally allocated on the heap). The fact that "strings" are actually pointers is a frequent source of errors, as is the explicit management of the storage containing the characters. We expect the availability of garbage collection to substantially change the normal programming style associated with these sequences, but did not wish to invalidate present Mesa programs using strings.

We have introduced the new built-in type TEXT for a (packed) sequence of characters. REF TEXT will generally replace most uses of STRING, which, however, will still be available in unchecked regions as a synonym for POINTER TO TEXT. To reduce signed/unsigned number problems, the maximum length of TEXT will be 2↑15.

"String literals" will be used to denote both STRING and REF TEXT constants with the READONLY attribute; context will be used to make the distinction (as with CARDINALS and INTEGERS).

Characters are not the only sequences that are useful; dynamic storage reclamation will make sequences even more useful, and safety will require that their use be carefully restricted. We have added a constructor for sequence types to the language. Conceptually, a sequence is a cross between an array (because its elements are all the same type, and can be indexed) and a variant record (because a tag indicates the structure—in this case, the length—of the balance of the object). Sequences will be sued for objects whose size is not bound until allocation; we have removed array descriptors from the checked language.

SEQUENCE decl OF type is a new union type constructor analogous to the variant part of variant records. The declaration must declare a tag of an index type, whose value will indicate the length of the sequence (and be used in bounds checking of accesses), and must be supplied as a parameter to NEW unless a default is given in the type constructor.

Example:

```
MyString: TYPE = REF MyText;
MyStringIndex: TYPE = [0 .. 1000);
MyText: TYPE = RECORD [
                    currentLength: MyStringIndex,
                    PACKED SEQUENCE maxLength: MyStringIndex OF CHARACTER];
outputString: MyString ← NEW MyText[500];
```

Exported Types

Exported types provide a type-safe mechanism for supporting an object-oriented programming style generally similar to that of Humus. (Further details are provided in a later section.)

Exported types may be used either *concretely* or *abstractly*. A concrete type is obtained from a particular module instance that exports the type. The exporting module instance is considered to be part of the type; thus, concrete types obtained from distinct implementations may be different types. An abstract (generic) type is obtained from a definitions module by specifying ANY; every abstract value implicitly carries a reference to an implementing module instance, as well as explicit data. Concrete values can always be widened to abstract values with the same type name declared in the same definitions module (or any definitions module that it extends); narrowing requires a run-time test for identity of implementing module instances.

Processes

Process handles offer the same problems as other potentially dangling pointers, but we wish to keep them in the safe subset. The implementation will be changed to include additional checking for erroneous situations (e.g., JOINing a process more than once). ABORT will be treated as a signal in the receiving process (with the additional property that the process is awakened to handle it, if it is waiting on a condition variable); it will cause termination of the process (after unwinding) if there is no handler for it in the aborted process.

GetCurrent and Detach are restricted to unchecked regions. The safe language construct DETACH P will have the semantics of Detach [FORK P].

> This implies that the decision of whether a process is to be detached must be made when it is originally created.

Unsafe Features

Although some of them have safe uses, the following Mesa language features may be used only in unchecked regions:

LOOPHOLE

UNCOUNTED, POINTER

Arithmetic to produce references

DESCRIPTOR

FREE

MACHINE CODE

TRANSFER WITH

RETURN WITH

STATE

Many procedures provided by the current (unchecked) Mesa run-time system have unsafe interfaces, and hence cannot be called from checked regions. Examples are UnNew, UnNewConfig, and SelfDestruct.

DELAYED BINDING

Type Values

We propose to add a new predefined type called TYPE. TYPEs are full-fledged values exactly like INTEGERs—there can be TYPE variables, parameters, results, fields, etc. The elementary TYPE constants are the PredefinedTypes listed in Appendix F of the Mesa manual, to which type TYPE itself is added. The primaries of type TYPE are type identifiers, optionally preceded by adjectives and qualified by field names. Type expressions are built up using the operators given under TypeConstructor in Appendix F of the Mesa Reference Manual. The section on syntax contains the modified grammar.

> 6C3 will probably generalize type primaries to standard Mesa primaries, including subscripting, qualification, procedure calls, conditionals, etc. (i.e., the same rules as for primaries of any other type, sans type-specific operations). Then typeId can be replaced by var. (The RELATIVE and Subrange alternatives may introduce ambiguities into the language; if so, we will need to resolve them.) In Mesa 6C2, we require that all the quantities in type constructors (both the types and the auxiliary quantities like interval bounds) be compile-time constants. Furthermore, it is our intention that in *all* versions of Cedar Mesa, the type expression appearing in a declaration be restricted to a compile-time constant, except for exported/imported types in interfaces, and whatever kind of polymorphism mechanisms we may invent.

In sections below we will introduce a few more additions to the type system for handling dynamically typed pointers.

Operations

We propose a language construct TYPE[x] which gives the type of x as an object of type TYPE. This computation can always be done at compile time, except that the final mapping to the run time representation must be done at bind or load time (see the next section). TYPE[x], however, must still obey "scope" restrictions: if x's type cannot be named in a scope (e.g., because it is private), then TYPE cannot be applied to x.

For Mesa 6C2, the only run-time operations we propose for types are assignment, comparison for equality, and conversion to an S-expression (which allows discovery of the entire internal structure of the type). Conversion to an S-expression is always safe (i.e., cannot violate compile-time scope restrictions), provided that the representation is in the form of ordinary records and not other objects of type TYPE, except at the terminal nodes. This operation is provided by a library package, not as a language construct.

Dynamically Typed Entities

We propose to add two kinds of dynamically typed or type-carrying (TC) entities to Mesa: TC references in 6C2 and TC procedures in 6C3. A TC reference is denoted by the type REF ANY.

Conformance of TC Entities

For any type T, REF T conforms freely to REF ANY. Hence, if pA: REF ANY and pT: REF T, the assignment pA ← pT is always legal and has the obvious effect. The usual rules for widening, etc. apply here. The opposite assignment pT ← pA requires an implicit narrowing; i.e., it is equivalent to pT ← NARROW[pA, REF T].

Use of TC Entities

We extend the WITH ... SELECT construct to allow the OpenItem to be a TC entity, and the Adjectives to be arbitrary TypeExpressions. Within each Choice of the SELECT, the variable in the OpenItem is treated as though it were of the selected type, e.g.,

```
ri: REF INTEGER;
pTC: REF ANY;

WITH p: = pTC SELECT FROM
        REF INTEGER => ri ← p;
        ENDCASE => ERROR;
```

The test being applied to the dynamic type of the TC entity is type equivalence (i.e., identical representation at run time). Ideally, the test should be free conformity: we consider the amount of work required to support this test too great for Mesa 6C2.

Alternatively, the programmer who wishes to treat failure of conformity as an exceptional condition may use NARROW[pTC] or NARROW[pTC, type]. This raises a signal RangeError if pTC is not of the appropriate type. Thus the above program could have been written:

```
ri: REF INTEGER;
pTC: REF ANY;

ri ← NARROW[pTC ! RangeError => ERROR];
                                -- target type for NARROW is REF INTEGER
```

Note that the above constructs are the only way to convert a TC entity back to a statically typed entity, which can be operated on by normal Mesa constructs.

If x is TC, then TYPE[x] is *not* the dynamic type of x, but is x's declared type (e.g., REF ANY). For TC objects, the construct DynamicType[x] gives the dynamic type.

Concrete and Abstract Types

In present Mesa, all types are concrete types, so the meaning of the "type of an entity" is relatively simple (although not entirely without pitfalls, as we noted above: sometimes T: TYPE = INTEGER should be considered the same as INTEGER, and sometimes it should not). However, once we introduce interfaces as objects in the language, an object may have both an interface (abstract) type and a concrete type. This is further complicated by object-oriented programming: an object may actually have several abstract types, namely its own interface type and those of its superclasses. In Mesa 6C2, TYPE[x] will return x's "most concrete" type.

Procedure Application

Current Mesa requires that any invocation of a procedure supply an explicit list of arguments and result locations: only the identity of the procedure itself is left variable. Cedar Mesa will relax these requirements in two ways: in 6C2 the arguments and results may be manipulated as records (corresponding to our earlier observation that they are just ordinary data objects), and 6C3 will allow "opaque" application in which nothing about the procedure's type is known at compile time and relatively little (beyond a compatibility check) is required at run time.

APPLY

APPLY Proc [ArgRec] will apply a procedure to a pre-constructed argument record. ArgRec must conform to the declared argument record type for Proc. The conformance rules for procedure arguments and results have been generalized to allow records in place of explicit lists, with the record conforming if all its fields do. I.e., if a value of a record's type (written as a constructor) would be an acceptable argument list, then the record is an acceptable argument to APPLY; if it would be an acceptable extractor, the value of APPLY may be assigned to it.

Lists

LIST OF is a new type constructor.

```
L: TYPE = LIST OF T;
```

is exactly equivalent to the (recursive) type definition

```
L: TYPE = REF N;
N: TYPE = RECORD[first: T, rest: L];
```

where N is a "private" name, except that lists are *not* "painted"; i.e., all LIST OF T types are equivalent.

If a, b, and c are of type T, the constructor

```
L[a, b, c]
```

will be equivalent to

```
NEW N ← [first: a, rest: NEW N ← [first: b, rest: NEW N ← [first: c, rest: NIL]]]
```

More generally, if t is of type T and lis is of type L,

```
L[t]
```

will be equivalent to

```
NEW N ← [first: t, rest: NIL]
```

and

```
L[t, lis]
```

will be equivalent to

```
NEW N ← [first: t, rest: lis]
```

(which is essentially Lisp's cons[t; lis]), and

```
L[t, "anything else"]
```

will be equivalent to

```
NEW N ← [first: t, rest: L["anything else"]].
```

As with other record constructors, the type may be omitted when implied by context.

No further additions to the type system or syntax are proposed for lists. Polymorphism for lists will be handled entirely by (and only to the extent of) the exported types mechanism, until the problem is solved for the language in general. (Note that the lack of polymorphic array procedures has not kept ARRAY from being a useful type constructor.)

EXPORTED TYPES AND OBJECTS

Properties of Objects

Since consideration of "object oriented" programming motivated many of the proposals in this section, we begin by considering some of the properties that characterize a satisfactory treatment of objects.

To check and compile client code that uses an object of a particular abstract type, four kinds of information are needed:

1) *Syntactically legal protocols (checked at compile time):* The legal operation names and arguments.

2) *Storage (computed at compile time):* How much storage to allocate for an instance of an object; if a garbage collector is to be able to trace the object, additional information to interpret this storage.

3) *Method lookup (at run time):* How to find the code for the operation, given the particular instance of the object representation stored in the client, and how to pass the storage representation to the method.

4) *Consistency checking:* How to ensure that the piece of code found actually agrees with the legal protocol ensured by 1.

Examples

Legal Protocols: Lisp record accesses (which can be general operations) are checked for legality by ensuring that the field name is defined for one and only one record. Message protocols in Smalltalk are unchecked at compile time. Humus objects use a specification of procedure types in the form of an interface declaration (definitions module), as does our object proposal.

Storage: Lisp stores all objects as typed pointers, so storage size is fixed and so is interpretation by the garbage collector. Smalltalk also uses a uniform representation scheme for all objects as did Humus. We are proposing a standard representation consisting of a [data, REF interface] pair and have arranged things so that the reference to the interface can often be implicit. The size of the data must be specified in the abstract type declaration, or else clients may only reference the object indirectly.

Method Access: Lisp uses the property list or function cell of the Litatom to find the method in a standard way. For records, this is done at compile time. Lookup uses a system-wide procedure and is independent of the object instance that the method is going to be applied to. In Smalltalk, a message name is constructed out of the message pieces in a standard way. The lookup method uses a system-wide procedure for finding a message name in a dictionary associated with the particular object. In Humus, an index into a record of procedures is compiled into the code, where the index number is computed by the compiler on the basis of information in a definitions module. Our scheme is similar; a definitions module implicitly defines an interface record of procedures, and an index into that record appears in the compiled code.

Consistency checking: In typed record accesses, Lisp explicitly checks the type of the object at each access; in untyped record accesses, it simply believes the programmer. In Smalltalk, the only consistency check is whether the message name actually appears in the dictionary of the specific object. In Humus, the system is forced to believe the programmer (through the use of a necessary LOOPHOLE at object creation time). In our proposal, the Mesa compiler and binder will ensure the type safety of the transaction.

Consistent Implementations

Cooperation among clients of an interface frequently requires the knowledge that different instances of an interface supply the same (but unknown) implementation. In current Mesa, it is not possible to specify this restriction within the clients. If different implementations are accidentally intermixed, the chances for confusion and error are substantial. We have added a feature to allow importers of multiple instances to specify whether or not they are to be treated as equivalent. Interface instances prefixed by ANY in the imports clause are allowed to have *non-standard implementations*. The binder/loader will ensure that all other instances of an interface are implemented by the same module(s), since this is considered to be the default case.

Distribution of implementations to clients is controlled by configuration descriptions. Suppose that FooClient imports two instances of Foo, and declares at least one of them to allow a non-standard implementation. The following statements in the configuration description language C/Mesa cause FooClient to nevertheless import two identical instances of Foo, as implemented by FooImpl:

```
FI: Foo ← FooImpl[];          -- FI is an instance of Foo
FooClient[FI, FI];
```

The following statements cause distinct implementations to be used:

```
FI1: Foo ← FooImpl[];         -- FI1 and FI2 are instances of Foo
FI2: Foo ← FooAltImpl[];
FooClient[FI1, FI2]
```

Clients may also use generic types for dealing with multiple instances (see below). This solution is unnecessarily general (and correspondingly more expensive) if it is known that the implementations are to be the same.

Exported Opaque Types

In current Mesa, it is somewhat inconvenient to make a module's clients independent of the representation of any type used in the interface. If one doesn't want to use any loopholes, he uses a PRIVATE declaration in the definitions file. While this makes the representation inaccessible, it still forces the clients to recompile every time the representation changes. To solve this problem, Cedar Mesa will allow types to be treated more like procedures—only information essential to a client is declared in the interface, while the complete definition is supplied by the implementing module(s).

We introduce *opaque types* as a new kind of interface component. The structure and representation of an opaque type are largely invisible to an importer of the interface in which it is declared. An exporter of an interface may supply a complete definition for opaque types in that interface; the types so supplied are called *concrete types* and are properties of an interface instance, not of the interface definition itself. All modules in a particular implementation that use the concrete form of such a type must export the *same* concrete type to the interface; however, different implementations may supply different concrete types.

Types that are references to opaque types are also considered opaque.

Since concrete types are supplied by implementations and opaque types are properties of instances of interfaces, types with the same name exported from two distinct instances of an interface are treated as distinct if either instance is declared to have a non-standard implementation.

The size of an opaque type must be known when compiling clients, if any operations are to be allowed. The default is for the size to be unknown, with no operations allowed. Such opaque types may be used in clients only as object types for references. In general, operations on the abstract type are provided by procedures declared in the same interface. If the type is declared in the interface with a size (in words), then allocation (including declaration), assignment (including

parameter-passing), and comparison for equality are supplied by the system, with the values being treated as uninterpreted bits; the concrete type supplied must require the specified number of words and may not be RC (since the compiler cannot know what reference-counting is required). If the interface is safe, the concrete type may not be PC, either.

Defining an Opaque Type

The following interface Foo introduces a type Fie, which refers to an opaque type Baz:

```
Foo: DEFINITIONS =
    BEGIN
    Baz: TYPE;
    Fie: TYPE = REF Baz;
    Create: PROC [INTEGER] RETURNS [Fie];
    Read: PROC [Fie] RETURNS [INTEGER];
    Munge: PROC [x: Fie, y: Fie];
    ...
    END
```

Consider the following skeletal module:

```
FooClient: PROGRAM IMPORTS
        F1: Foo,                -- a specific, but unknown implementation
        F2: ANY Foo =           -- another (non-standard) specific implementation
    BEGIN
    x, y: F1.Fie;               -- will refer to F1's representation of Fie
    z: F2.Fie;                  -- will refer to F2's representation of Fie
    i: INTEGER;
    ...
    END
```

Within the client, the types of F1.Create, F1.Read and F1.Munge are respectively

```
PROC [INTEGER] RETURNS [F1.Fie]
PROC [F1.Fie] RETURNS [INTEGER]
PROC [x: F1.Fie, y: F1.Fie]
```

and similarly for F2. Thus the following statements are legal:

```
y ← F1.Create[5];
x ← y;
i ← F1.Read[x];
F2.Munge[z, z];
z ← NARROW[F1.Create[5]]; -- signals RangeError if F1 and F2 have different implementations
```

while the following are not (because they potentially mix different concrete types):

```
z ← F1.Create[5];
x ← z;
i ← F1.Read[z];
i ← F2.Read[x];
F1.Munge[z, x];
F2.Munge[z, x];
```

An exporter of Foo may provide a definition of the type Baz and (as in current Mesa) implementations of procedures declared in Foo; for example:

```
FooImpl: PROGRAM EXPORTS Foo =
    BEGIN
    Baz: PUBLIC TYPE = RECORD [a, b: INTEGER];
        -- Do not be misled by the "PUBLIC"; only the name Baz is public.
    Create: PUBLIC PROC [x: INTEGER] RETURNS [t: Foo.Fie] =
        BEGIN
        t ← (NEW Baz ← [a: x/7, b: x MOD 7]);
        RETURN
        END;
    Read: PUBLIC PROC [x: Foo.Fie] RETURNS [INTEGER] =
        BEGIN RETURN [x.a*7+x.b] END;
    Munge: PROC [x, y: Foo.Fie] =
        BEGIN x.a ← y.a END;
    ...
    END
```

Note that, within FooImpl, Baz is a concrete type, and thus so is Foo.Fie.

Guaranteeing Consistency

Type safety in the use of opaque types is based largely upon the following properties of Mesa:

> The compiler's type checking guarantees that all items exported by a module to a particular interface are type-consistent; in particular, all exported operations that involve an opaque type expect the same concrete type.
>
> Within the language, interfaces are immutable; there is no way of updating the value of an interface or any of its components.

Operations on interfaces must be defined with care to maintain this type safety. Currently, the Mesa binder can merge instances of an interface, as specified by the PLUS and THEN operators of C/Mesa, to produce a new instance. The binder must check that the concrete types supplied for the opaque types are identical before doing the merge. Any similar operations for manipulating interfaces that are added to the Mesa language must be implemented with similar checks.

Multimodule implementations of an interface containing an opaque type are straightforward. All the modules that use the type concretely share a private interface definition containing the concrete declaration of the type, and all export that type (to ensure checking). By the above conventions, the exported (partial) interfaces can be merged without conflict.

A Notational Convention

We adopt a convention that allows "object-oriented" notation to be used in conjunction with opaque types. Let T be a type declared in an interface definition D. If I is some instance of D and if the type of x is I.T (or REF I.T), then x.Op[args] is interpreted as I.Op[x, args]. (The expression preceding the dot becomes either the first argument (positional notation) or the argument with name SELF (keyword notation)). With this convention, the legal statements above can be rewritten as follows:

```
y ← Fl.Create[5];          -- no rewrite possible; Create does not take a Fie
i ← x.Read[];
z.Munge[z];
```

Generic Types

When there are multiple implementations of some abstract type coexisting within the same system, a client often wants to intermix representations freely, without restricting the values of a given variable to a single concrete type. This has been accomplished through the use of various kinds of "objects" in the past; the most notable effort in Mesa was the set of object conventions adopted by Humus.

The basic idea in our proposal is that a client may declare an opaque type to be *generic* (or abstract) by referring to it using just the interface name (rather than the name of a specific instance of that interface). If D is a definitions module defining an interface with opaque type T, we denote the generic type by ANY D.T. A variable with that generic type can store a value having any concrete type implementing T. The representation of a generic value includes a reference to the instance of the interface that provides the definition of, and operations upon, that particular concrete type.

> Note that if D.T is not prefixed by ANY, it denotes whatever concrete type is exported by the "standard" implementation, i.e., it is the same as any of the I.T's.

We also adopt the following convention for method lookup (requirement (3)): if an operation Op is referred to as a component of a generic variable x, the implementation of Op will be chosen on the basis of the concrete representation of x. Thus x.Op[args] is interpreted as I.Op[x, args], where I is the interface record appropriate for the concrete implementation of the present value of x.

There is an automatic coercion from an opaque type to the corresponding generic type. E.g., the type D.T conforms (but not freely) to the type ANY D.T. Such assignment is a widening operation. Type safety in the use of generic types (requirement (4)) is based upon the type safety of the opaque types.

Generic types introduce another form of delayed binding. The concrete type of a value with a generic type is determined dynamically. That type is itself a component of the interface associated with the value. If the type of the generic variable x is ANY D.T, the type of the concrete representation is denoted by x.T. The latter is a TypeExpression but not a type constant (it can be used in narrowing operations, etc., but not in declarations).

The concrete type can be discovered using the extension of the WITH ... SELECT construct introduced in the delayed binding section. Alternatively, a generic value can be narrowed to a concrete one; the narrowing succeeds if the representation of the value has the specified concrete type. (Note that an opaque type evaluates to a concrete type for the purpose of type discrimination.)

The following declaration can be added to FooClient; it declares variables that can store any representation of a Fie:

```
u, v: ANY Foo.Fie;
```

Then the following statements are valid:

```
x ← F1.Create[5];
z ← F2.Create[8];
u ← x;                  -- a widening
i ← x.Read[];           -- F1.Read will be used
i ← u.Read[];           -- the appropriate Read will be used
```

The following examples illustrate some forms involving implicit narrowing and their explicit equivalents:

```
x ← u;                          -- the implementation of u is unknown.
x ← NARROW[u];                  -- explicit alternative
z ← NARROW[u, F2.Fie];          -- a more explicit form
-- which is equivalent to
WITH w : = u SELECT FROM
        F2.Fie => z ← w;
        ENDCASE => SIGNAL RangeError;

-- note that both parameters of Munge must have the same concrete type
u.Munge[z];                     -- requires a narrowing; we can't be sure u's type is F2.Fie
NARROW[u, F2.Fie].Munge[z];     -- possibly generates a signal
u.Munge[NARROW[z, u.Fie]];      -- possibly generates a signal

u.Munge[v];                     -- requires a narrowing; u and v may have different concrete types
u.Munge[NARROW[v, u.Fie]];      -- possibly generates a signal
```

Notice that, if F1 and F2 are in fact the same implementation, the assignment "z ← NARROW[x]" will succeed, even though the compiler will disallow assignment of x to z directly.

Multiple Implementor Interaction

When an interface provides a procedure such as Munge that takes two or more parameters with opaque types, the possibility of an implementor dealing with objects created by other implementors exists. To indicate a willingness to do so, the prefix ANY may be used to declare parameters and variables. For example, a redeclaration of Munge in Foo as

```
Munge: PROC [x: Fie, y: ANY Fie];
```

allows calls such as

```
u.Munge[z];     -- a widening of z
u.Munge[v];
```

The implementor is expected to deal with the variability of the second parameter. The implementation of Munge might be rewritten as follows:

```
Munge: PUBLIC PROC [x: Fie, y: ANY Foo.Fie] =
        BEGIN
        x.a ← WITH v : = y SELECT FROM
                Fie => v.a,
                ENDCASE => v.Read[]/7;
        END;
```

Summary

Types whose definition is given explicitly are concrete. If an interface declares a type without giving the definition, the type is opaque to importers of the interface (except, of course, to implementers that declare it concretely). Clients of an interface may use an opaque type in any of three ways:

The default case is for all instances of the type to have a standard (but unknown) implementation. This is helpful in "information hiding" and in breaking compilation dependencies among modules.

By declaring multiple instances of the interface, some with the attribute ANY, the client can

use multiple, distinct implementations of what is logically the same interface. The type-checking rules will ensure that these implementations are never intermixed.

By declaring generic variables (using ANY and the definitions module—rather than instance—name), the client can freely intermix implementations. The select statement and the object-oriented notation for procedure invocation can be used to cause the procedures appropriate to a particular implementation to be applied only to objects with its concrete type.

Subclasses

Simula and Smalltalk provide a "subclassing" mechanism. This allows common properties of a group of related "classes" of objects to be declared in a single "superclass" of which they are all "subclasses." Its advantages include code sharing, more coherent structures for many programs, and consistency of changes when the program is modified.

Suppose the definitions module DefsA declares operations P and Q. Then the definitions module DefsB can declare a new interface which provides operations P, Q, R, and S (with P and Q declared as in DefsA) by saying EXTENDS DefsA and just declaring R and S. This has the following consequences:

A set of modules that implement DefsB must define P, Q, R and S.

If T is an opaque type declared in DefsA, a value with type ANY DefsB.T may be assigned to a variable with type ANY DefsA.T.

If IA and IB are instances of DefsA and DefsB respectively, IB.T is assignment compatible with IA.T if both instances are standard, i.e., if there is no possibility that the different interface instances supplied different concrete types.

The binder will accept an implementation of DefsB wherever an implementation of DefsA is required.

This mechanism can be used for subclasses (DefsB is a subclass of DefsA). Note that this approach even works for multiple superclasses—all that is required is to have more than one embedded EXTENDS clause, and a rule for handling the conflict that arises when more than one superclass declares the same operation name. In Mesa 6C2, such a conflict will be an error.

An interface being extended may declare an opaque type T. When this happens, implementors of the extended interface have two choices: define T directly or treat T opaquely, by importing an instance of the interface being extended and "passing on" its definition of T. (Similar remarks apply to procedures declared in the interface being extended.)

In the first case, the list of items in the interface being extended is essentially concatenated with the list of items in the extension. The result is a "flat" structure, and the exporter directly supplies the values in the entire flattened interface. This situation is likely to arise when the concrete representation is specific to the extended interface. A familiar example is the extension of Stream to DiskStream, as suggested by the following:

```
Stream: DEFINITIONS =
    BEGIN
    Handle: TYPE = REF StreamData;
    StreamData: TYPE;
    Get: PROC [Handle] RETURNS [CHARACTER];
    EndOf: PROC [Handle] RETURNS [BOOLEAN];
    ...
```

```
        END
DiskStream: DEFINITIONS
            EXTENDS Stream =
        BEGIN
        Create: PROC [fileName: STRING] RETURNS [Handle];
        ...
        SetIndex: PROC [Handle, INTEGER];
        ...
        END
```

An exporter of DiskStream provides not only a definition of StreamRecord and additional operations (such as Create and SetIndex) but also implementations of Get, EndOf, etc. that may differ from the implementations of corresponding operations for other representations of streams. The extension thus does not result in the sharing of code between implementors; its important consequence is that the types DiskStreamInstance.Handle and ANY DiskStream.Handle are assignment compatible with the type ANY Stream.Handle.

In the second case, both the interface and its implementation are being extended. We add some notation to allow convenient and efficient "passing through" of the value of an imported interface to an exported interface that extends it. Consider, for example, an extension of Foo defined as follows:

```
FooExt: DEFINITIONS
            EXTENDS Foo =
        BEGIN
        Crunch: PROC [Fie, INTEGER];
        ...
        END
```

An exporter of FooExt has the following form:

```
FooExtImpl: PROGRAM
            IMPORTS F: Foo EXPORTS FooExt, F = -- values in F are "passed through"
        BEGIN
        Crunch: PUBLIC PROC [x: F.Fie, i: INTEGER] =
                BEGIN ... END;
                ...
        END
```

Note that FooExtImpl does not itself define the exported type Fie; that definition and the implementations of the operations named in Foo are provided by the implementor of the instance imported by F. Thus Fie is treated like any other opaque type within the body of FooExtImpl.

GARBAGE COLLECTION INVARIANT

A programmer who writes in unchecked Cedar Mesa is responsible for ensuring that nothing his program does can cause inaccuracies in the view of memory held by the allocator and garbage collector. Aside from direct modifications to storage allocation data structures, the only actions that will affect this view of memory are *assignments* (assignment statements and certain machine language statements that change the contents of memory), NEW, and FREE statements. The rules below are sufficient (but not strictly necessary) to ensure that these operations do not lead to an invalid allocation or collection.

BASIC DEFINITIONS

At any point in time each storage location is in one of the following three states:

1. *Accessible*: the location is in use to store a data item that is in a global frame, in a local frame in the stack of some process, or can be reached by following references from such a frame variable. Equivalently, a location is accessible if it can be addressed by a safe left-hand-side expression. (We will also say that a *reference* is accessible if it is stored in an accessible location.)

2. *Allocated:* the location was part of the storage allocated by a NEW for a pointer, and it has not subsequently been released by a FREE statement.

3. *Unavailable*: neither of the above.

Both accessible and allocated memory locations are available for use by the program. The difference is that accessible memory is managed by the garbage collector, while allocated memory is not. A large part of the responsibility of the unsafe-language programmer is to see that the types assigned to accessible locations are unambiguous with respect to the position of counted references.

GLOBAL INVARIANT

The following invariant must be preserved by all atomic actions. (Actions that satisfy the rules given below will automatically preserve the invariant, so its presentation here is essentially a comment).

1. Memory is unambiguously typed with respect to references.

2. If there is an accessible reference to a memory location, then the garbage collector has a non-zero reference count for that location.

RULES FOR ASSIGNMENT

It must be possible to express the assignment by a statement of the form

lhs := rhs

where lhs and rhs are expressions that satisfy the following rules:

1. The storage for lhs is either accessible or allocated.
Effect: assignments can only modify parts of memory that are available to the program.

2. The lhs expression does not involve a loophole to or from a type that contains references.
Effect: reference counts kept by the collector remain accurate.

3. The lhs and rhs have types that are reference compatible, i.e., their memory layouts agree on the positions and types of references. [Obviously equal types are reference compatible. Other compatiblities may be implied by the language definition.]
Effect: the assignment maintains type consistency.

4. All references in the rhs are accessible.
Effect: the assignment does not cause any new memory locations to become accessible. This is necessary to maintain type consistency.

A VAR parameter may appear on the lhs of an assignment in a procedure. Thus, any VAR argument should satisfy rules 1-3 above (using the type of the corresponding formal parameter as the type of the rhs in rule 3).

Rules 2-4 are restrictions on assignments to references. Since the collector ignores pointers, it is unaffected by the values stored in them, and there are no corresponding rules for assignments to pointers. However, if a pointer is used on the left-hand-side of an assignment, the programmer must have some way of knowing that the memory it points to is available. For example, if the pointer was obtained by loopholing a reference, the programmer must be sure that the storage is still accessible, i.e., that there is still some accessible reference to it.

RULES FOR FREE(P)

1. p refers to allocated storage locations.

2. p satisfies the rules for a VAR parameter (see above).

3. (Optional, but recommended) p is the last reference to this storage. (This is recommended because it makes it easier to check that rule 1 for assignment is satisfied.)

RULES FOR NEW

None.

RULES FOR PROVIDING SAFE INTERFACES

To (honestly) declare an interface SAFE, the programmer must ensure that no operations of checked programs can cause violation of the garbage collection invariant; in particular, no possible set of parameters, or checked use of the results, can cause storage to be mistyped.

The most important consequence is that if pointer values are passed across the interface (after having been loopholed into some other type), it must be possible to guarantee that the rules for FREE still hold, e.g.,

> because the values passed out to checked regions are never re-loopholed into pointers, or
>
> because storage referred to by such pointers is never freed.

NOTES FOR IMPLEMENTATION

This section consists of a sequence of extracts from the 5C1 document that are relevant to the implementation of 6C2, but are not actually part of the language design. It is expected that this section will be restructured and grow considerably as the implementation progresses.

SAFE LANGUAGE

Persistent Closures

We intend an efficient implementation for the overwhelmingly common case where frames obey a stack discipline, and can be freed on procedure return, but this should be transparent to the programmer.

There is a representation problem, since an inner procedure is currently packed into a single word with a trick that does not comply with our desire not to have pointers into the middle of allocation units. We propose to solve this problem (and may also remove the current system restriction on the number of global frames) by changing the representation of procedures. The exact nature of this change has not been worked out yet: there is clearly a workable default, which is simply a record containing a reference to a frame and a small number for the entry point index.

Here's a first shot at how to deal with persistent frames (ignoring ports and processes for now):

> When a procedure value for P is constructed, mark the frames of all of P's statically enclosing procedures "persistent."
>
> When control returns from a frame which is marked "persistent," make the frame collectible instead of reclaiming it. If the called procedure took VAR parameters, mark the current (calling) frame "persistent."

Bounds and NIL Checking

Most NIL checks can be performed at low overhead by the paging hardware. The entire page (perhaps even several pages) to which NIL's address points will be reserved for this purpose. The representation of NIL will uniformly be a zero of appropriate length; both widening and narrowing operations must check for NIL and convert it appropriately.

Safe Features

Some detailed implementation consideration is needed to ensure that ports and processes are as safe as we think they are.

DELAYED BINDING

A Preliminary Note on Unique Ids

The Cedar runtime environment includes several kinds of objects whose true identifications require many bits, but which are used in so many places that a representation using few bits is highly desirable. Examples of such objects include *atoms* (uniquely stored strings), *interfaces* (representing compiled Mesa interfaces, identified by <file name, time stamp>), and *types* (representing Mesa types, identified by <file name, time stamp, compiler symbol table location>); others will undoubtedly arise in the future. We propose that the internal representation of each of these objects be a short id (one word) which is mapped into the true representation by an internal hash table. This approach is feasible only because the number of such objects that actually can be mentioned in the execution environment at any given time is much less than the potential (or even actual) number of such objects in the universe.

The following trick produces an efficient bidirectional mapping between short ids and true identifications. Let there be a hash table H, containing only the short ids (not the actual data), and an indexed table T, containing some encoding of the actual data. (For example, for the atom table, T would in principle be an ARRAY OF REF READONLY TEXT.) Given a short id i, the corresponding data are T[i]. To map from data d to the corresponding short id, do a hash lookup in H, as follows:

```
probe: CARDINAL ← FirstProbe[d];

WHILE H[probe] # 0 DO
    IF T[H[probe]] = d THEN RETURN[H[probe]];
        -- actually use TextEqual, or whatever
    probe ← NextProbe[probe, d];
    ENDLOOP;
RETURN[0];
```

The extra level of indirection (storing the indices rather than the actual data in H) allows us to return the short id H[probe] if the entry is found.

Dynamic typing

For Mesa 6C2, the only requirement that we place on the runtime representation for types is that different types be distinguishable, hence we have few constraints on the choice of representation. To minimize space, we assign a short unique identifier to each value of type TYPE; the compiler's representation of types is converted to these identifiers when an object module is loaded.

Since our present machines handle only fixed-size objects efficiently, we have chosen to associate dynamic type information only with references (which are fixed-size). We offer two ways of associating a type with a reference:

For zones where it is likely that many objects of the same type will be allocated, we associate the type with an entire block (quantum) of storage.

For zones where it is objects of various types may share a quantum, we store the type with the object.

Type Values

Representation

The concrete representation (i.e., short id form) of type TYPE is:

```
TYPE: TYPE = RECORD[index: PRIVATE CARDINAL];
```

The abstract identification of a TYPE is actually a pair containing the unique id of the module in which it was defined (including the time stamp), and the symbol table index of its definition; as described in the earlier section on unique ids, there is a runtime structure that maps such a pair into a unique short id. For programs containing type constants, the Binder (or Loader) must do the conversion and put the result into the global frame, exactly as it does for string constants.

We take the view that types considered equivalent by Mesa have identical run time representations, e.g., T: TYPE = INTEGER is equal to INTEGER. This may complicate the hash lookup algorithm considerably, depending on how the work is divided between the compiler and the loader,but if we had taken the opposite view, the WITH ... SELECT construct is essentially useless.

Dynamically Typed Entities

Representation

The representation of a REF ANY is the same as that of any other REF, namely, a long pointer.

The current type of the object addressed by a dynamically typed reference pAny is found according to the type of the zone in which its referent is located by:

```
pAddress: LONG CARDINAL = LOOPHOLE[pAny];
pType: TYPE;

pType ← WITH zone: SystemZoneMap[pAddress/SystemQuantumSize] SELECT FROM
        -- SystemZoneMap is an ARRAY [0..AddressSpaceSize/SystemQuantumSize) OF Zone known to the system
    prefix => LOOPHOLE[pAddress-1, REF TYPE]↑,
        -- type is in preceding word
    quantum => zone.objectType,
        -- zone is actually an internal "subzone" object which represents a type-homogeneous set of quanta in a quantum-mapped zone
    ENDCASE => ERROR NotDynamicallyTyped[pAny];
```

DynamicType is actually a call through the system dispatch vector SD that performs the foregoing computation.

EXPORTED TYPES

Exported Opaque Types

Implementation Notes

This part of our proposal amounts to a rather small compiler change. We simply require that type checking be done in a slightly different way. An opaque type looks unique to an importer; it cannot be confused with any other type. It looks like a unifiable variable to an exporter; for purposes of checking he can substitute any single type he wishes for it.

In the representation of an interface, we reserve space for recording the concrete type exported for each opaque type. Conceptually, this identification is a unique type identifier (<interface: <string, time stamp>, entry: symbol table index>). We expect to map these unique ids into short ids as discussed above.

The consequences for the binder (and its runtime analogs) are somewhat greater. It must compute and merge collections of short unique ids. In addition, the binder must check operations upon interfaces; two interface instances may be merged (using PLUS or THEN) only if the values of any opaque types are equivalent.

Generic Types

Implementation Notes

A generic value is actually a pair with the type

RECORD [ops: REF RECORD[PROC ...], value: REF]

If x has an opaque type from interface F1, and u has the corresponding generic type, the widening assignment u ← x is actually u ← [@F1, x].

A type discrimination tests the identity of the corresponding type component of F1 (which will be a short unique id). A procedure call such as x.Read[] is an abbreviation for x.ops.Read[x.value].

Subclasses

Implementation Notes

The compiler embeds (a reference to) the interface record defined by DefsA at a known offset within the interface record defined by DefsB. The assignment of an ANY DefsB.T to an ANY DefsA.T requires just an adjustment of the component of the representation designating the interface record.

COLLECTED GRAMMAR

Reference [Ivy]<Lampson>MesaGrammar.bravo

This section was produced by minimal editing of Butler Lampson's Mesa 6 Grammar document; we are indebted to Butler for both the style and the bulk of the content.

The intention of this grammar and associated restrictions is to document the Mesa 6C2 syntax (to be accepted by the compiler). The notes following rules are semantic restrictions; a considerable amount of the detail in the manual's syntax has been moved into them, so as to simplify the grammar.

Notation:

Parentheses are for grouping.

Item | item means choose one.

?item means zero or one occurrences of item.

{item}sep means one or more occurrences of item separated by sep. The sep is optional. ?{item}sep means zero or more.

Terminals are punctuation other than (){}?| or quoted or in SMALL CAPS.

Intentional changes from Butler's grammar (i.e., syntax specific to 6C2) are in **bold face**. Underlined stuff is unsafe. All of this is comments.

The order is: module, type, expression, and top-down within these.

module ::= ?(directory ;) id : transferTC ?(EXTENDS ids)
?(IMPORTS { ?(id :) id },) ?(EXPORTS ids) ?(SHARES ids) = block .
The transferTC must be PROGRAM, DEFINITIONS or MONITOR.

ids ::= {id},

directory ::= DIRECTORY {id : FROM stringLiteral ?(USING [?ids])},

block ::= attributes (BEGIN body END | "{" body "}")

body ::= ?(open ;) ?(enable ;) ?{decl | stat}; ?; ?(finish ?;)
The enable can't use "!"
The decls must all precede the stats, and must all have ids.

open ::= (OPEN | WITH) {?(id : ?=) exp},

decl ::= ?({id ?position}, :) type
Position only in the fields of a recordTC (or unionTC?).

finish ::= (EXITS | REPEAT) {choice};
REPEAT iff the body is a loop body (crock).

position ::= "(" exp ?(: exp .. exp) ")"
Only for machine-dependent stuff.

stat ::= exp | control | block | loop
An exp in stat context must not produce a result.

control ::= CONTINUE | EXIT | (GOTO | GO TO) id | LOOP | NULL | RETRY | STOP | (RETURN | RESUME) ?group | var ← STATE
The id after a GOTO is in a separate scope from other names in the block (crock?).

loop ::= ?iterator ?((WHILE | UNTIL) exp) DO body ENDLOOP

iterator ::= THROUGH exp |
FOR id ?DECREASING IN exp | FOR id ← exp , exp
The IN and THROUGH exps must be subranges.

type ::= attributes (typeId | builtInType | typeCons) ?initialization
A field init is called a *default*.

typeId ::= ?{id | ANY} id ?{ . id | group}
The prefixed ids are adjectives.

attributes ::= ?{PUBLIC | PRIVATE | READONLY | ENTRY | INTERNAL | MONITORED | MACHINE DEPENDENT | id RELATIVE | LONG | ORDERED | PACKED | INLINE | RESIDENT | COMPUTED | OVERLAID | MACHINE CODE | CHECKED | UNCHECKED | SAFE | UNSAFE | UNCOUNTED | VAR}
VAR only in parameter lists.
Many restrictions on which types and blocks can actually have which of these, and their ordering.

builtInType ::= INTEGER | CARDINAL | REAL | UNSPECIFIED | TYPE
TYPE only in a body's decl, with = initialization.

initialization ::= (← | =) (exp ?("|" NULL) | block | type | CODE | [exp])
= only in a body. | NULL only in defaults, CODE only for signals, [] only for strings.
Block only for procedures. Type only for = init of types.

typeCons ::= enumerationTC | * | subrange |
recordTC | unionTC | arrayTC | refTC | transferTC | storageTC
* allowed only for the tag of a unionTC. Meaning: make up a nice type.

enumerationTC ::= "{" {id ?position}, "}"
BOOLEAN={FALSE, TRUE}; CHARACTER={...}
The position must be just an integer here.

recordTC ::= RECORD fields

fields ::= [?{decl ?position},] | NULL
All must be named or all unnamed. No defaults if unnamed.
Unnamed and unionTC => nothing else (crock). Any unionTC must come last.
NULL is a (useless) synonym for [] in a unionTC only.
Position only in recordTC (or unionTC?).

unionTC ::= (SELECT decl FROM {ids => fields}, ?, ENDCASE | SEQUENCE decl OF type)
At most one in a recordTC.
Only one id on tag, present iff not COMPUTED or OVERLAID. No initialization (?).

arrayTC ::= ((ARRAY ?type) OF | LIST OF | DESCRIPTOR FOR) type
Index type must be simple, may be elided for a descriptor only.
DESCRIPTOR requires array type.

refTC ::= (POINTER ?subrange TO | REF) ?(type | ANY)
No id on the subrange.
There is a crock for FRAME, not indicated here.

transferTC ::= (PROCEDURE | PROC | PORT | PROCESS | SIGNAL | ERROR | PROGRAM | MONITOR | DEFINITIONS)
?fields ?(RETURNS fields) ?(LOCKS primary ?(USING decl))
No parameters for process or definitions. No returns for monitor. One id, no init for USING.

storageTC ::= (BASE | ZONE) ?group

exp ::= primary | prefixOp exp | exp infixOp exp | subrange | var ← exp | funnyAppl var ?group | if | select | NULL
No syntactic distinction between exp and a subset of stat.
Precedence is noted in bold in the operator rules. All operators associate to the left.
Assignment to an extractor produces no result, hence must appear in a stat context.
Subrange only after IN. NULL only in defaults and groups.

primary ::= var | literal | builtIn group | DESCRIPTOR [exp] | DESCRIPTOR [exp , exp ?(, type)]

var ::= id | "(" exp ")" | var ↑ | var . id | group | var group
An exp in this context must produce a result.

group ::= [?{?(id :) exp}, ?(enable)]
In an extractor the exps must be vars.
No enable in constructor, extractor, indexing, builtIn, loophole. Only the "!" enable.
If any elements are named, all must be.

enable ::= ! {choice}; | ENABLE choice | ENABLE BEGIN {choice}; ?; END

builtIn ::= MIN | MAX | ABS | LENGTH | BASE | SUCC | PRED | SIZE | FIRST | LAST | ALL | LONG | TYPE | LOOPHOLE | PUN | NARROW

prefixOp ::= @ (8) | - (7) | ~ | NOT (3)
infixOp ::= * | / | MOD (6) | + | - (5) | relOp (4) | AND (2) | OR (1)
relOp ::= ?(~ | NOT) (# | = | < | <= | > | >= | IN)
subrange ::= ?typeId ("[" | "(") exp .. exp ("]" | ")")

funnyAppl ::= FORK | JOIN | SIGNAL | ERROR | NEW | APPLY | START | RESTART | WAIT | NOTIFY | BROADCAST | RETURN WITH ERROR | TRANSFER WITH | RETURN WITH | FREE | ABORT | DETACH
The second and third line funnyAppls produce no result.
ERROR without arguments is short for ERROR ERROR.
NOTIFY and BROADCAST cannot have an enable.

if ::= IF exp THEN stat ?(ELSE stat)
In an expression, the ELSE is required and the stats must be exps.

select ::= ?open SELECT exp FROM {choice}sep ?sep ENDCASE ?(=> stat)
The open must use WITH, and have only one item.
In an expression, the stat following ENDCASE is required and the stats must be exps.
The separator sep is "," in exp, ";" otherwise.

choice ::= {condition}, => stat

condition ::= exp | relOp exp | FINISHED | ANY | type
SELECT requires exp or relationTail.
WITH requires either type constants (for disciminating TC entities) or else ids called adjectives whose scope is the tag type of the variant openItem.
EXITS and REPEAT require ids called labels (or FINISHED). These double as declarations, with scope only the GOTOs of the containing body (not the whole body).
Enables require exp or ANY, not id or ANY as you might think.

Inter-Office Memorandum

To	Cedar filing, Cedar language	Date	February 20, 1980
From	B. W. Lampson	Location	CSL, Palo Alto
Subject	System modelling, February 1980	File	[Ivy]<Lampson>ModelsNow.press

XEROX

Last edited by BWL, February 20, 1980, 10:44 AM

This memo contains a proposal for a system modelling facility which is consistent with the long-range goals of Cedar, but can reasonably be implemented now and used with the current Mesa system. Eric Schmidt is embarking on an implementation. Comments are solicited on two questions:

> Is it right? I.e., is it consistent with the long-range goals of Cedar? Does it address all the important issues, and address them correctly?
>
> Is it feasible now? Can this proposal, or the subset of it indicated in the comments on current implementation which appear here and there, be implemented in time to be useful for Cedar development?

Note: this is not a user's manual.

Acknowledgements

This approach to system description originated in the design of the Mesa binder, which I did with Charles Simonyi, John Wick and Rich Johnsson. I am indebted to the members of an SDD working group on consistent compilation for many fruitful discussions. Jim Horning and Andrew Birrell also helped to shape these ideas during meetings of the Cedar filing design group. Eric Schmidt has pointed out a number of problems.

1. Framework

A *system model* is a piece of source text which contains a complete description of a Cedar system; i.e., all the information needed to construct the system is present. A model may have (formal) *parameters*, however, and in that case (actual) *arguments* must be supplied to completely define a system.

There is code to support system models (called the modeller). It:

> Invokes the compiler, binder etc. to construct the system described by a particular model, and tries to keep track of when this work has already been done and need not be repeated;
>
> Provides convenient ways of updating a model to reflect new versions of the component modules or models (would you believe *modelules*, or *modulels*) which come along during the development of a system;
>
> Supplies the necessary interfaces to the file systems which actually store the information, copies data to local disks when needed, and so forth.

Remark: One of the reasons this has been hard to get straight is that it requires understanding polymorphism (type parameters) to some extent, because Mesa actually has polymorphism. It only shows up in definitions modules, was not recognized very clearly, and is not done right. Some of the problems came out more clearly as a result of the desire for exported types, which are another form of polymorphism. In fact, global frames are also exported types. Furthermore, any PRIVATE type probably should be exported, though the current language provides no protection against mixing up two implementations of such a type. This didn't seem important because it doesn't affect the integrity of the built-in types. ∎

Remark: Another persistent source of confusion is that details of the implementation have often been taken as conceptually fundamental. In this exposition, such details are relegated to their proper place (I hope). ∎

The description in this memo is couched in terms of Mesa. However, the entire scheme is applicable to any collection of source files and programs for converting them into a running system (*translators*), provided only that all the parameterization can be specified when the translators are invoked. I expect that in the long run the modeler will be usable for other such applications, e.g., font shuffling and hardware design automation, with only minor changes to the existing translators for those applications.

2. Units

A particular system model with no formal parameters contains (in principle) all the source text from which the system is generated; i.e., it is like a card deck for the entire system. In fact, of course, this text is stored on a number of files scattered throughout the file system. To take account of this, we introduce the notion of a *unit*, which is a piece of source text with no free identifiers (but perhaps with formal parameters). Because it has no free identifiers, a unit can stand alone, and need not be part of a larger structure in order to be meaningful. A unit is *immutable*; its contents never change. Of course, it is possible to use the contents of a unit as a basis for constructing a new unit; this process is often called *editing*. A model itself is a unit, and so is a Mesa module. Smaller units, such as procedures or even statements, are also possible (but not in the initial implementation).

The immutability of units is an essential aspect of the scheme, since it allows a name to stand unambiguously for the contents of a unit, and ensures that no property of the named object can change (an edited object would have a new name). Among other things, this means that information about the object which is cached, e.g., in a data base maintained by the modeler, cannot become invalid and hence never needs to be checked. The price for this, of course, is lots of new names, and hence it is essential to have convenient ways of updating a model to refer to a new object; see section 6 below.

Each unit has an *identifier*, which is unique throughout the entire universe of discourse. The identifier is a string, and is composed of *parts* which are alphanumeric strings, separated by dots or bangs; e.g., Compiler.Parser.RepairError!16. Uniqueness is guaranteed by a decentralized registration procedure. A *naming authority* for the identifier i is denoted $i+$ or $i!d\uparrow$, where d is a string of digits called a *version number.* An identifier j *descends* from i if

$i=p+$ and p is a proper prefix of j, or

$i=p!d\uparrow$ and $p!e$ is a prefix of j for any $e>d$.

Two identifiers *conflict* if they are equal, or either descends from the other. An authority contains a *log* which is a list of the identifiers it has issued; it may issue any identifier which descends from it and does not conflict with a identifier already in the log.

In the initial implementation units correspond to files. Since our file systems cannot ensure the immutability of a unit, or any fixed correspondence between name and contents, it seems desirable for the modeller to do this. We therefore append to each file its identifier (to fix the correspondence) and a checksum of the file contents including the identifier (so that immutability can be checked), in the form <<<identifier: checksum>>> *(or something like that).* If this is done by looking for <<<...>>> near the end of a file and replacing the ..., the client can embed this construct

in a comment for all languages and file formats we currently have (except pictures). The intention is that the file system name of a unit will be the same as its identifier, possibly with a prefixed directory.

Note that there is nothing wrong with having multiple copies of a unit in the file system, since all are identical and immutable (of course multiple copies of a naming authority are bad). In fact, we *expect* multiple copies to occur, when units are copied to local disks.

Obviously it is necessary to have some way of knowing where to look in the file system for a particular unit, since exhaustive search is impractical. This could be done with a special data base, but it is easier to include this information in the system model, as described below, and we will start that way. Note that such file system pointers are simply hints; if they are wrong, the modeller may not be able to proceed, but it will never use the wrong unit. Presumably it could also ask the user where to look in case of difficulty.

I don't yet have a position on whether we should implement naming authorities, or do that manually for the moment. I think that on Juniper it wouldn't be a big deal.

3. Modelling language

The following grammar describes the modeling language. It is taken from the simplest abstract grammar for Cedar/Mesa in the current draft of the Cedar abstract machine description. Hence, if no mistakes have been made, it is a subset of (abstract) Cedar/Mesa, except for the stuff in bold, which was added specifically for modelling. Details are open to negotiation; in particular, (in)consistency with the concrete syntax has not been carefully considered.

Notation for the grammar:
Parentheses are for grouping. Operator precedence is | (lowest), ?, }.
Item | item means choose one.
?item means zero or one items.
{item}sep means one or more items separated by sep (which may be empty).
Terminals are punctuation other than (){}?| and --, or quoted, or in SMALL CAPS.
Subscripts and from -- to end of rule are comments, not part of the grammar.

exp ::=	{ ?{id ,} part}; | {part}, |
	SELECT call FROM [{exp_1 => exp_2};] -- exp_2 whose exp_1=primary
part ::=	id : call
call ::=	primary group | primary == call
primary ::=	literal | id | **unitId** | "(" exp ")" | primary . id |
	TYPE **?id** | STRING | PROC ?group RETURNS ?group
group ::=	[?exp]
unitId ::=	**@ location uiPart | unitId ! {digit} | unitId . uiPart**
location ::=	**?([uiPart]) ?{uiPart}>**
uiPart ::=	**id | id ↑**

Operators are left-associative.

We may need integers and enumerations here at some point, e.g., to state compiler parameters? If not, we can probably flush primary.id from primary.

Problems:

We still need PLUS and THEN.

An expression is a list of named parts, each with a type and a value. Since there is no assignment in this language, the value must be supplied on the spot, except when the exp defines a parameter or result type. The type may be omitted, in which case it is inferred from the value. For historical

reasons, either comma or semicolon may separate the parts; if semicolon is used, *a*, *b*: *V* is short for *a*: *V*; *B*: *V*. The identifiers of the exp denote the corresponding calls throughout the exp. Calls are used to define and apply procedures; they are explained in the next section. *T*==*V* is short for *T*[*V*]; *T* must be a type.

The unitId stands for the text of that unit. In a unitId, a naked identifier *x* denotes the STRING literal "*x*", and *x*↑ denotes the value of the identifier *x*, which must be a STRING (and is probably a parameter). The only alternative I can find for this somewhat peculiar syntax is to require the literals to be quoted explicitly. That seems very clumsy, since a literal is the normal case inside a unitId.

The id following TYPE allows a crude check that the type (defs module) being obtained is the one expected: a value of type TYPE *X* is produced by a module which starts *X*: DEFINITIONS. The basic job of the type system is to ensure that an argument has the expected properties; this is the one place where that job cannot be done properly, since it is precisely the statement of those properties which is being supplied. The identifier serves as an informal abbreviation of the missing information. If it is omitted in a part (i.e., [... *X*: TYPE ...]), the part's id (i.e., *X*) is used instead.

We write *A*→*R* in this document as short for PROC *A* RETURNS *R*, or for PROC [*A*] RETURNS [*R*] if *A* and *R* have only one part.

4. Parameters

In a call the primary must be a proc or a type. In the former case the meaning is to bind the arguments in the call to the parameters of the proc, and then evaluate the body of the proc. The types of the arguments must match the types of the parameters. This is ordinary procedure call. Note that the modeling language is purely applicative; it does not have assignment. Hence parameter binding is just substitution of the arguments for the parameters in the body. The use of explicit NEW and UNNEW operations, and other modification of interfaces at runtime, makes the computation which produces the running system non-applicative. This is not necessarily bad, but there is a price in loss of clarity. If the modelling language is also to be used for describing the state of the running system, there will have to be a way to prettyprint that state as a model. This is an issue for the debugger.

The result of applying a type is to coerce the argument exp to have that type; thus CARDINAL[3] or CARDINAL==3 is a value of type CARDINAL. In the modelling language, this is only interesting when the type is a proc type.

The result of applying a proc type, $(A \rightarrow R)[i_1: e_1, ..., i_n: e_n]$, is a procedure whose body is the argument expression, with parameters which are the parts of *A*. For each part *i*:*T* of *R*, there must be a matching i_j and e_j must have type *T*. For example, consider

PROC [*a*: ..., *b*: ...] RETURNS [*x*: ..., *y*: ...]==(*v*: ..., *w*: ..., *x*: ..., *y*: ...)

The body declares *v*, *w*, *x* and *y*, and may refer to all these and to *a* and *b*. The result is the values of *x* and *y*. Thus λ*x.e* in the untyped lambda calculus would be written (*x*:*T*→*y*:*U*)[*e*] in this notation, for some suitable *T* and *U*.

A program module has a proc type, with parameters specified in the DIRECTORY (type parameters) and IMPORTS (interface record parameters) clauses, and results specified in the EXPORTS clause. The imported and exported interfaces have types which are the values of defs modules; hence their values are just records of procedures and (pointers to) variables, plus constant components which are not stored. With exported types, this is no longer true, since an exported type is a part which can vary from one instance to another; however, it still isn't necessary to store the typeid, though it may be convenient to do so. A definitions module with no DIRECTORY has type TYPE *X*; if it has a DIRECTORY its type is [...]→*X*: TYPE *X*. Thus

```
DIRECTORY A: FROM "Afile", B: FROM "Bfile";
X: DEFINITIONS = { ... };
```

has the type [*A*: TYPE *A*, *B*: TYPE *B*]→*X*: TYPE *X*; we could have elided the tags and written this [*A*: TYPE, *B*: TYPE]→*X*: TYPE. Note that "Afile" and "Bfile" do not appear; they are the *arguments* for the parameters *A* and *B* of *X*, and as such do not belong in the definition of *X*.

As explained above, the *X* in TYPE *X* is not really necessary. Without it, however, a TYPE parameter would accept any DEFS as an argument, and this could lead to considerable confusion if the wrong module was supplied by mistake. If *X* is specified, then it must match the *X* of the argument; this check is likely to catch most such errors.

Of course models can be parameterized also. In fact, normally a model will be a call of the form

(PROC ?group RETURNS ?group) [...].

There is no restriction on the type of a parameter. In particular, a parameter may have type TYPE; a type parameter may then appear in the result type of the proc type. This is not a frill; otherwise it would not be possible to treat a module as a unit with only parameters and no free identifiers. For example, suppose we have a program module *Ximp* which exports (implements) a definitions module

Xdefs=*Xtag*: DEFINITIONS={...}.

The model for this is:

X: TYPE *Xtag* = @*Xdefs*!14;
xi: *X* = @*Ximp*!7[*X*];

Here @*Ximp*!7 has the type *X*:TYPE *Xtag*→*xi*:*X*. From now on we will use *X* for *Xtag*, and elide it where possible. Thus our model becomes

X: TYPE = @*Xdefs*!14;
xi: *X* = @*Ximp*!7[*X*];

and *Ximp* has type *X*:TYPE→*xi*:*X*. Note: most current Mesa users would have *Xdefs* where we have *X* here, but this is strictly a matter of convention; Pilot, for example, follows the opposite rule and has *X* where we have *Xdefs*. In practice the unitId will likely be the same (or have a part which is the same) as the type tag, but we keep them different to emphasize the different kinds of objects involved.

It is also possible to supply arguments for a subset of the parameters; the result is then a proc with the remaining parameters as its parameters, the same result, and a body which is the body of the original proc with the supplied arguments substituted for the corresponding parameters. Thus

(([*x*:*T*, *y*: *V*]→*U*)[...*y*...*y*...])[*y*: *e*] = (*x*:*T*→*U*)[...*e*...*e*...]

Such a partial call is called a *projection. There is an unfortunate conflict between this notation and the Mesa notation for defaulting arguments.*

Projections are interesting because the compiler produces them: a module cannot be compiled until all its type (defs) parameters are supplied (except the opaque (exported) types). Thus, if we extend the example above with another interface *Y* and a module which imports *X* and exports *Y*, we get

X: TYPE = @*Xdefs*!14;
Y: TYPE = @*Ydefs*!1;
xi: *X* = @*Ximp*!7[*X*];
yb: (*xi*:*X*→*yi*:*Y*) = @*Yimp*!3[*X*, *Y*];
yi: *Y* = *yb*[*xi*];

Here @*Yimp*!7 is a unit with the type [*Y*:TYPE, *X*:TYPE, *xi*:*X*]→*yi*:*Y*, and *yb* corresponds to the BCD: it still needs a value of type *X* to produce a usable interface. Note: it is wrong to say that *yb is* the BCD; it is just a projection of *Yimp*. The model has no concept of BCD.

If we treated procedures, rather than modules, as the smallest units, we would see that a global frame is just like any other interface (observation due to Andrew Birrell). That is, its type must be known to the compiler, and the instance can be supplied later. Of course the implementation is significantly different at the moment.

The resemblance of this to C/Mesa is certainly not accidental. The difference is that in C/Mesa there is no explicit description of, or control over, the type parameterization; which argument is supplied for a type parameter is entirely determined by what happens to be on the disk under that file name.

5. Abbreviations

The example above shows that writing out all the details is rather wordy. Things are improved by the following coercion:

$x \Rightarrow x[a_1, ..., a_n]$ whenever x: $[a_1: T_1, ..., a_n: T_n] \rightarrow U$

This might be called "strong deproceduring." With it, we can omit all the argument lists in the example.

We also note that the types do not generally have to be written, since they can be inferred from what follows the = sign. If what follows is a procedure, the ambiguity about how many parameters should be suplied is resolved by supplying all of them. Hence if a projection is required, its type must be stated explicitly. Fortunately, in many cases there is no reason to mention the projection explicitly, especially if it is applied only once.

With these abbreviations we can write the example more concisely:

X: = @*Xdefs*!14;
Y: = @*Ydefs*!1;
xi: = @*Ximp*!7;
yi: = @*Yimp*!3;

Both omissions of course reduce the value of the model for documentation. Also, the modeller will probably want to keep the types around, since otherwise it must examine all the files. This suggests that the text of the model with all the information should be the unit, and it should be possible to get it printed with less detail. It should also be possible tell the modeller to examine the files and update the type information. In fact, the types are exactly the information which the current include checker recovers from the BCDs. Using the BCDs for this purpose is contrary to the modelling philosophy, however, which is that only the sources are real. It also has the drawback that everything has to compile before the information is available. Since only the directory, imports and exports lists need to be examined, it is no big deal to get the types from the source.

6. Editing and noticing units

When a unit is changed, the result is a new unit with a different identifier. Typically the change is done as part of the development of some system, which is represented by a model. For the change to have any effect on the system, it is necessary to produce a new model which refers to the new unit; then the system described by this new model can be built. We call this *noticing* the unit (the term is borrowed from the Interlisp file package).

Notice: [*M*: Model; *old*: Unit; *new*: Unit]→*M'*: Model

searches *M* and, recursively, any models to which it refers, for occurrences of *old*; each such occurrence is replaced by *new*. Of course, when this is done in a model, the result is a new model, references to which are in turn replaced. The resulting model refers only to *new*. When a new model is made during this process, it has to have a new identifier. If the old identifier has the form *n*!*d*, the modeller obtains *n*!*d*↑ (the SDD Librarian calls this checkout), and uses *n*!(*d*+1) for the new one. Otherwise it asks the user what to do. *Doubtless other reasonable strategies are possible. A nice one might be to append the user's initials* !1, *and then to count up the version.*

Noticing should be done automatically when a file is edited, so that the single user developing a program by editing modules will hardly be aware of the model. To this end, the modeller should be told when a file is changed by the IME editor, and hopefully also by Bravo (perhaps using the measurement log). There should also be a way to ask it to check all the write dates of files

mentioned in a model, in case the incremental process gets confused. It seems highly desirable to package the modeler into either IME or the compiler, in order to avoid an extra context switch. Furthermore, the modeler should keep the current model (and any submodels which it had to expand because there wasn't a single BCD) in memory as a graph, so that the file doesn't have to be read and parsed on every edit.

We should also have *Add* and *Remove* operations;

Add[*M*: Model; *new*: Unit; {*before, after*}; *old*: Unit]

inserts *new* before or after the (unique) occurrence of *old* in *M*. It would be nicer to do these operations by editing a picture of the model, but that is a longer-range feature. With *Notice*, these should take care of all ordinary changes to the model during straightforward program development by a single person.

If this kind of development proceeds independently along two paths, both starting with *M*, one often wants to merge the results back into a single new model.

Merge[*Base*: Model; *M*1: Model; *M*2: Model]→Model

traces out the three models depth-first. At each level:

If	*Add to result*
Base.p= *M*1.*p*= *M*2.*p*	*Base.p*
Base.p= *M*1/2.*p*≠ *M*2/1.*p*	*M*2/1.*p*
Base.p= *M*1/2.*p*, no *M*2/1.*p*	leave *p* out
no *Base.p* or *M*1/2.*p*	*M*2/1.*p*
Base.p≠ *M*1.*p*≠ *M*2.*p*, all models	*Merge*[*Base.p*:, *M*1.*p*, *M*2.*p*]
ELSE	error, or ask what to do.

Of course, there is no guarantee that the resulting thing makes any sense, but it does seem to correspond to current practice. If we don't go in for checkout, it will also be necessary to choose the right name for a unit being newly inserted.

A nice feature of *Notice* and *Merge* is that they still seem to do the right thing even if the analysis is pushed below the level of models and modules, down to individual procedures or further, to compound statements (but I am not advocating this for now).

7. Compiling and binding

In order to actually build a system, the modeller must get everything compiled and bound. Thus, for each call of a unit, it must collect BCDs for all the type (defs) parameters, and values for all the parameters corresponding to compiler switches, and call the compiler to produce a BCD. Then it must construct a C/Mesa .config file for the model (perhaps for each sub-model) and call the binder.

Obviously, we don't want this done from scratch after every tiny change. Hence the modeller must keep some kind of data base which tells it what BCDs already exist. Logically this has a very simple form: a projection $U[U_1, ..., U_1]$ always produces the same BCD (this is why all compiler switches, including the compiler version, which is not strictly a switch, must be included among the parameters; the deproceduring coercion will keep us from having to write these things). Hence the data base simply lists pairs <projection, BCD file>.

Questions to be resolved:

How to represent the data base. Eventually, use the Cedar facilities. In the short term if they aren't ready, perhaps keep a file with name derived from the unit name which lists all the BCDs and associated parameters. Or this file could lump stuff for a bunch of units, as long as it doesn't get too big. Or whatever.

When should old BCDs be removed? This is strictly a space-time tradeoff, since they can always be rebuilt from the sources.

Do we want to make any effort to make the grain of dependency finer, as Desoto did? Probably not.

We consider the details of how modules get loaded and started to be outside the province of the modeller. They are part of the runtime system.

Inter-Office Memorandum

To	Cedar Interest	Date	March 31, 1980
From	Jim Morris	Location	Palo Alto
Subject	The Cedar Project	File	[Ivy]<Cedardocs>Status1Q80.press

XEROX

Background and Goals

Xerox plans to develop and market office systems of increasing power and internal complexity. It will be necessary to develop technologies enabling the solution of broad ranges of office problems with systems that are composed of separable, tractable components with standard interfaces, yet are perceived by their users as customized and integrated.

To facilitate these plans, CSL produces prototype systems that provide interactive, personal computing services. Some of these systems are short-lived experiments to test novel ideas, and some are solid systems that are used by many people. It is important that some of these systems receive wide use, beyond the walls of CSL and PARC; this has proved to be a powerful means of technology transfer as well as a way for us to learn about large, distributed systems. Besides office prototypes we also develop things such as compilers, operating system components, program verifiers, etc. that are of general interest to computer scientists.

At any given time there are around sixty professionals in CSL who might be programming. Although they are all sophisticated programmers, they have a wide range of goals and styles: system programming, mathematics, hardware design, artificial intelligence, etc. Most of them are permanent employees who can be expected to invest effort in learning how to use a programming system, but an appreciable fraction are visiting scientists or students who can be productive only if they can assimilate the basic system quickly.

The Cedar project is an effort by CSL to produce a new programming environment. The main goal of this new environment is to increase programmer productivity, specifically the productivity of the programmers in CSL over the next several years, beginning soon. This improvement will come from three main sources: A language system that takes more responsibility for certain programming tasks, programming tools that make program development and debugging faster, and a package library that allows programmers to build upon one another's work. To a large extent these facilities are enabled by the increased capablities of the D0 and Dorado computers. Given the precedent of the Alto system, it is plausible that the D-machines and Cedar will be used by other groups someday.

Acceptance of this specific goal and its immediacy has made us conservative in our designs. We have restricted ourselves to those ideas which can be understood and put to use by the intended users in a timely fashion. While it is our business as computer scientists to strive for new and revolutionary ideas, they are not required for Cedar. Indeed, employing the users of Cedar as guinea pigs for such ideas will tend to decrease their programming productivity in the time frame of interest.

History

A few years ago the Computer Science Laboratory began the process of changing its basic hardware from the Alto to the D0, a larger, Mesa-oriented machine, and the Dorado, a very powerful machine. As these machines began to materialize several people, most notably Peter Deutsch, saw the opportunity to make a large stride in our approach to software. Specifically, the increased capacity of the Dorado made it possible to contemplate using much higher level programming languages and more powerful programming tools on a personal computer.

In early 1978 a group began discussing various design alternatives and developed a general consensus on what features a new programming environment should have [1]. The general idea was that we wanted to combine the best features of the Lisp/Smalltalk environments—automatic storage management, delayed binding, fast turn-around, and an integrated environment—with those of the Mesa language—static checking, explicit interfaces, and efficient use of machine resources.

Later that year a larger group met and explored the possibilities of undertaking the effort starting from a Lisp base or a Mesa base [2]. Eventually the choice was made to start with Mesa.

In 1979 a large, distributed design effort was begun on many different topics: user facilities, display packages, language changes, communications, management of large programs, filing, etc. This effort was coordinated by a steering committee consisting of Peter Deutsch, Jim Horning, and Butler Lampson, and involved a substantial fraction of the laboratory. An external design review was held in September. The organization of the implementation efforts began in the late summer of 1979 and the programming of certain things started in the Fall.

The Cedar project makes heavy use of certain core software developed by SDD. As starting points for Cedar we are using their implementations of the microcode for D0's, the Mesa language, and the Pilot operating system. The interim Mesa environment described below uses the Tools environment heavily. Finally, Ed Satterthwaite, a member of the Mesa group in SDD, has been a key contributor to the Cedar project from the beginning.

Core Projects

The core of the Cedar project consists of four levels rising from the raw hardware of the D machines to the system used by the programmer: the microcode, the Kernel operating system, the Mesa language, and User Facilities. The people listed at the end of a paragraph are the ones doing the work; the asterisk indicates the current coordinator of the effort.

Moving to D-machines

A significant improvement in our programming environment comes from changing our hardware from Altos to D0's and Dorados. Not only are these machines larger than the Alto, they support a fully general virtual memory, are better suited to running Mesa, and allow us to use the Pilot operating system. Pilot brings with it a carefully designed virtual memory, a good file system organization, and a better implementation of the Mesa process mechanism. Making this move will require work to adapt SDD's microcode and operating system to our needs.

The microcode for the Dorado and D0, like the hardware itself, is not actually part of the Cedar project, but is so crucial that we include it for completeness. It implements the Mesa instruction set, performs low-level I/O functions, and implements certain time-critical operations associated with automatic storage allocation. It is essential that it present the same interface for the Dorado and D0.

> *Status:* Versions of Alto Mesa run on both machines
>
> *Plans:* Basic support for the Amargosa Pilot instruction set on CSL D0's will be provided when work on the Kernel begins in April. General performance improvements, especially relating to long pointer operations will be undertaken. Support for the Kernel on the Dorado will follow; it represents significantly more work. Special instructions to speed up automatic storage allocation will be available in the Fall. A re-work of the procedure call mechanism will be needed to support the new design for interfaces and objects. [Fiala, Taft, Thacker*]

The Kernel operating system provides the basic facilities of an operating system: processes, virtual memory, and a local file system. It is essentially the Pilot operating system as amended for Cedar's needs [3]. The Kernel does not address the management directories or distributed files very extensively; see Filing Facilities below.

> *Status:* Pilot has been demonstrated running on a CSL D0. This required adjusting to certain device variations.
>
> *Plans:* The Kernel interface described [3] will be available for use on D0's and Dorados in July. [Birrell, Gifford, Levin*, Schroeder]

Consolidation of Programming Languages

For several years, programmers at PARC and elsewhere have tended to fall into two groups: systems programmers who use compiled, machine-oriented languages (BCPL or Mesa) and programmer who use higher level languages (Lisp, Smalltalk). Locally, the first group produces systems that run on the Alto and receive widespread use by non-programmers. The second group produces prototypes which contain many new ideas, but cannot be used widely because they do not run on Altos or run too slowly for a non-programmer's patience. Although this diversity in programming systems has its virtues, it has some drawbacks:

> The communication of ideas between these two groups is impeded by non-substantive but highly loaded issues such as the syntax of the respective programming languages.
>
> Maintenance of multiple languages in a changing environment requires too much work.
>
> It is hard to produce an integrated system if the only common denominator is the hardware or file structure.

Aside from its technical objectives, Cedar is an attempt to improve this communication without compromising the essential styles of programming and system development of these two schools. There are many factors, technical and non-technical, affecting this objective; and we have not fully come to grips with all of them.

Having made the decision to start with Mesa and perceiving our design goal to be roughly the least upper bound of Mesa and Lisp, we singled out automatic storage management, delayed binding of type information, and the ability to manipulate programs as data structures as the most important additions. The specific and detailed design for language changes may be found in the Cedar Mesa 6C2 document [4]. An abstract machine definition is being developed to define a convenient interface for debugging and program manipulation. A primer for Cedar Mesa is being written by Bill McKeeman.

> *Status:* The garbage collector, manager of run-time types, and finalization procedures have been implemented by Rovner and Deutsch. A pre-release version of the Mesa 6 compiler, done by Satterthwaite, has been run on D0's.
>
> *Plans*:
> Phase 0: Mesa 6, SDD's alpha test release (May)
> Phase 1: Syntax changes for Cedar, Garbage Collection, Safe Language, Run-time types (July),
> Phase 2: Abstract machine and factored compiler made available to user facilities (Sept)
> Phase 3: More language features (sequences, generic types, interface records, persistent closures, var parameters, relative refs, etc.) (December)
> Later: Compacting storage allocator
> [Atkinson, Horning*, Rovner, Satterthwaite]

An integrated system of tools

A striking advantage of Lisp and Smalltalk over Mesa is that they have integrated, very powerful programming environments; and Mesa is a compiler, linker, debugger, and a handful of run-time routines. Lisp provides quick interaction while Mesa provides more thorough static checking at a more stately pace. We hope to create an environment for Mesa programming that provides tools in the spirit of InterLisp and Smalltalk. Of course, there are intrinsic trade-offs between flexiblity and static checking; we seek satisfactory compromises. At a minimum we hope to learn what is intrinsic and what is merely stylistic.

The User Facilities consist of editing and debugging tools incorporated into an integrated framework based upon some general purpose display and window packages. Some of the current thinking in this area may be found in various memos [5, 6, 7].

> *Status:* Some preliminary programming of a window package has been done.
>
> *Plans:* Basic text documents will be provided employing metaphors such as windows, desktops, and viewboxes. These are built on top of Warnock's display package as maintained by Wyatt. A general method of capturing and interpreting input (keys, mouse, etc.) will be provided. Basic debugging tools (break-points, value, program, and context display, etc.) will be built upon the abstract machine interface. These will be presented in a metaphor called the User's Debugging Panel. General executive functions (login, logout, file management) will be provided. These facilities should be available by the end of 1980.

[Crowther, Mitchell*, Paxton, Swinehart, Teitelman]

Allied Projects

There are some other basic system building activities in Cedar, but they are more open-ended than the core projects, and likely to outlive the Cedar project *per se.*

System Models

This effort is directed at the problem of programming in the large: how to manage a large, evolving system with multiple versions. A new language (resembling a subset of Mesa) has been developed to describe how configurations of modules are put together. [8] An implementation that operates by generating command files and configurations is currently underway. [Lampson*, Schmidt]

Data Bases

The immediate goal is to provide programmers with a package that permits the rapid storage and retrieval of tuples in a data base. The system uses Juniper as its filing medium. An initial release

for this system [9] is planned for May. Some applications will then be tried; system models may be one of them. [Brown*, Cattell, Suzuki]

Filing Facilities

Thus far this problem is being investigated by a large committee. The long range goal is to develop a single, integrated file system which spans personal work stations and file servers in a graceful way. Its design [10] includes an attempt to use the work stations' disks as caches rather than as permanent repositories of information. Its only short term objective is to provide a local directory system for the initial release of Cedar. [Brown*, Gifford, Kolling, Levin, Mitchell*, Schroeder, Sturgis, Taft]

The Interim Mesa Environment (IME)

IME was developed by Birrell and Cattell as scaffolding for Cedar. The Tools enviroment is used to provide an editor and debugger for Mesa programs. The larger memory of XM Altos, D0's, and Dorados is exploited in various ways. [Rovner*, Pier, Haugeland]

Problems, Short-falls, and Opportunities

Disparate machine capabilities

It is been estimated that the Dorado has five times the processing power of a D0, although the difference depends greatly on teh benchmark. During the early discussion stages, it was assumed that Cedar was to run primarily on Dorados. Since then we have recognized an increased importance of providing a basic system that runs on D0's as well because it does not seem likely that Dorados will become widespread enough to support an large community of users by themselves. Certain applications may require a Dorado to run reasonably, but the basic system must support the production of prototype systems for the D0. The difference between these two machines complicates many design considerations. On the other hand it also provides an impetus to explore distributed computing more vigorously. The value in off-loading computations from one Alto to others has never been obvious, but using a Dorado to perform certain compute-bound operations for D0's may make sense.

Pickled values

Since Cedar is a one-language environment we have the opportunity to extend the Mesa language type system to facilitate inter-machine communication as carried out via the Ethernet, file storage, and data bases. Earlier work on remote procedure calls addressed itself briefly to the question, but

there has not been much thinking about it since then. We need a scheme that permits a Mesa value to be converted to a permanent (pickled) form suitable for transmission to another machine or data base and subsequent incorporation into a running Mesa program. If this scheme worked for *any* Mesa value, including procedures and frames, it might prove useful as a sort of compacting garbage collector (but this may be a red herring).

Persistent Environment

It is nice to be able to save the state of a running environment in ferrous oxide from day to day. InterLisp, Smalltalk, and JaM all provide this facility; so there is no lack of models for what we want. Providing a persistent environment calls for the language's run-time system to have a robustness not provided by the current Mesa system. One needs to worry about smashed structures, space leaks, and fragmentation. These problems have been addressed on a application-by-application basis in the past, but now the system must take more responsibility. The compacting garbage collector is crucial in this regard.

Performance Analysis Tools

We need some good tools for understanding the performance of running programs, but not much effort has been devoted to them so far. The general philosophy of getting the functionality right and then improving the performance is not credible if the second step cannot be accomplished in a straightforward way. Gene McDaniel has recently been doing some good work in this area.

Sharing of programs and packages

One of the major arguments put forth for static checking, explicit interfaces, modularity, etc. is that they make the sharing and maintenance of programs easier. Also, removing concern for storage management should make many interfaces simpler and more robust. At the moment, however, we do not have a good collection of packages.

A major problem we have not solved is the management of the information about packages—documentation, bug reports, plans, etc. Paul Rovner, Rick Cattell, and Warren Teitelman have indicated interests in these problems.

The work underway so far on specific packages is in the area of displays, data bases, and caching. There is a long list of other useful packages to be developed (message transmission, forms, history lists, direct access to disk, dictionary service, teleconferencing, audio, direct access to network, string processing, small data base manager, ...). We hope that activity picks up after the basic system emerges.

Remote procedure calls

The ability to communicate between computers in a simple and natural way is the goal of the remote procedure call mechanism. It enables experimentation with various distributed computing regimes. It is described in a memo [11] and eventually will be addressed by the group working on the Kernel.

References

All the files mentioned are on [ivy]<cedardocs>

1. Peter Deutsch, First Report from Programming Working Group, July 1978, file: per>report>per.press

2. Peter Deutsch, Report from second Programming Environment Working Group, December 1978, file: pe2>pe2.press

3. Andrew Birrell, Roy Levin, Cedar Operating System Kernel, December 1979, file: kernel>kernel.press

4. Jim Horning, Cedar Mesa—Version 6C2, February 1980, file: lang>6c2>design.press

5. Dan Swinehart, A design for Documents and References in Cedar, October 1979, file: user>drafts>*.memo

6. Bill Paxton, Warren Teitelman, Viewboxes, Documents and Windows in Cedar, November 1979, file: user>viewboxes.press

7. Jim Mitchell, Minutes of User-Facilities Group, February 1980, file: user>15Feb80.minutes

8. Butler Lampson, System Modelling, February 1980, file: models>modelsnow.press

9. Mark Brown, Cedar Data Base Concepts and Facilities; Phase 1 implementation, March 1980, file: database>C+F.press

10. David Gifford, Cedar File System Proposed Concepts and Facilities, March 1980, file: files>pcf.press

11. Andrew Birrel, Cedar Communications, May 1979, comm>proposals.press

Laurel Message

From: ROVNER.PA

Date: 22 April 1980 11:30 pm PST (Tuesday)

Subject: 6C2 Atom Amendment

To: CSL↑, Cedar-Interest↑

XEROX

Date: 7 April 1980 2:52 pm PST (Monday)
From: Horning
Subject: 6C2 Atom Amendment
To: CSL↑, Cedar-Interest↑

Yes, Virginia, there will be atoms in Cedar Mesa. In fact, they have a rather high priority that was not reflected in the 6C2 document simply because the details of the proposal weren't settled in time. In general, you can assume that atoms have the properties that you know and love from your Lisp programming. More specifically, the following amendments to the 6C2 document of February 15 have been adopted by the language features design committee. Many thanks to Warren Teitelman for making this proposal gel.

PAGE 5: LISTS PARAGRAPH: REPLACE BY

Lists and Atoms

We have added a new type constructor, LIST OF, to declare singly linked lists whose elements are all of the same type. The record constructor notation is extended to construct list values in contexts that require them. ATOM is a new built in reference type, with properties similar to those of atoms in Lisp.

PAGE 21: NEW SECTION FOLLOWING LISTS SECTION

Atoms

Atom is a new built in reference type. The record constructor notation is extended so that ATOM[s] creates an ATOM value from a REF TEXT, STRING, or enumeration type value. Each atom value has a .pname component, for the atom's "print name," which is a REF READONLY TEXT. Space will also be reserved in the atom value to provide efficient access to property lists in the Atoms package.

The basic properties of atoms are given by the identities

ATOM[refText].pname↑ = refText↑

and

ATOM[atom.pname] = atom

These identities imply that atoms are equal iff their print names are equal as text;

hence, upper and lower case matter.

If context implies the type ATOM, s may be written for ATOM[s].

For any legal Mesa identifier, id, $id denotes the same atom constant as ATOM["id"]. Also note that

$NIL = ATOM["NIL"] = NIL = NIL[ATOM]

If id is a constant identifier in an enumeration type T, and e is an expression whose value is equal to id,

ATOM[e] = $id

and

NARROW[ATOM[e], T] = id

If the context demands an enumeration type, an atom value will be implicitly narrowed to that type, and RangeError will be signalled if the atom's print name is not one of the identifiers of the type.

PAGE 38:

Add | ATOM to list of alternatives for builtInType.

Inter-Office Memorandum

To *ADDRESSEE* Date April 2, 1980

From Andrew Birrell Location Palo Alto

Subject Processes in Cedar Organization CSL

XEROX

File [Ivy]<Birrell>Processes.bravo

Introduction

A number of changes should be made to Mesa processes to support Cedar, particularly to enable them to be used in safe language modules, and to provide facilities for the garbage collector. This memo outlines these changes.

Safe Language

There are several facilities in Mesa related to processes which, as they presently stand, would be unsafe. Here, "unsafe" means that a program which uses these facilities could not be guaranteed to allow the correct functioning of the garbage collector. Much of the unsafeness derives from the fact the PSB's are a fixed re-usable resource not subject to storage management by the garbage collector. Thus a client may improperly present a process handle which has been re-used, and which now represents some process other than that which the client intends, possibly even being a system process. Changing PSB's to be managed in a safer way would be too large an upheaval, so we present ways to make the facilities safe in the presence of re-used PSB's.

FORK and JOIN: The JOIN construct returns a value, whose type is specified by the type of the process handle which is being joined. If the process had in fact previously been joined (or detached), and its PSB had subsequently been re-used, then there may be a type mis-match between the value returned by the process and the value expected by the JOIN. This must be prevented, and the only possibility seems to be a dynamic check that the types match. This would require that when a process is FORK'ed in a safe module, its return type is recorded somewhere, and this is checked by software at the site of the JOIN. In any unsafe module, we cannot guarantee to have available a representation of the type returned by a process. This leads us to have two variants of FORK: a process FORK'ed in an unsafe module has its PSB marked as originating in an unsafe module; JOIN'ing to such a process is allowed only in an unsafe module. If we FORK in a safe module, the compiler generates code allowing us to record the result type of the FORK'ed process. We can implement this by an auxiliary table indexed by process handle: extensions to the PSB are not needed. To allow this table to be non-resident, we would require a single bit in a PSB to indicate the absence of a known result type. We are left with the possibility that the client may try to JOIN to a process whose PSB has been re-used for a system process. This is type-safe, but will

cause the system to malfunction. However, the system processes that we are concerned with here are those which are used to support the safe language; if we require that these processes were FORK'ed in unsafe modules then no problem arises. We also require that the implementation detect any attempt to JOIN to a detached process (easy).

Detach: The only objection to this is that the client may detach a process that has been re-used as a system process. This would be prevented if DETACH was a language construct (not a procedure call, as at present) and was allowed only at the FORK site, performed as an atomic action before the client process could terminate.

ABORT: We must prevent the client from aborting a system process. This is achieed by considering ABORT to be unsafe.

SetPriority: The only objection to this facility is that if the client sets his priority too high, he may prevent various system processes from executing. This is probably no worse than sitting in an infinite loop or allocationg all of the available memory.

Facilities for the Garbage Collector

The garbage collector occasionally needs to look at all the stack frames which are currently active, in a manner which provides it with a snapshot of the values referenced from these frames. It is interested only in client processes, and only interested in frames of "safe" modules. The major problem here is in providing the snapshot; it would also be nice if the evaluation stacks were in a clean state. Thus the garbage collector needs a facility allowing it to stop all the interesting processes. On the other hand, it must not stop those system processes which will be needed during the execution of the relevant code in the garbage collector, and it must not stop processes which are reacting to real-time stimuli. To handle this, we propose that there be two states for a process: "stoppable" and "unstoppable" (recorded by a bit in the PSB). FORK in an unsafe module creates an unstoppable process; FORK in a safe module creates a stoppable process. Any process can (as an unsafe action) declare itself to be unstoppable, and any unstoppable process can declare itself to be stoppable. Critical system processes and those driving real-time devices would typically be unstoppable (and therefore unsafe). When the garbage collector wishes to scan all the stack frames in the safe world, it asks the kernel to stop all stoppable processes. Our requirement is to ensure that the process machinery can achieve this in some finite time. It does so as follows. The state of the kernel is set to "stop stoppable processes". In this state, whenever a stoppable process becomes ready it is placed on the "stopped" queue instead of the "ready" queue. Thus, when a stoppable process is freed from a WAIT or from being enqueued on a monitor lock, is moves onto the stopped queue instead of the ready queue. When the kernel is in this state any process that declares itself stoppable is transferred onto the stopped queue, and any stoppable process executing an Xfer is transferred to the stopped queue. We are now left with processes which are in substantial iterative loops not containing any WAIT, nor any Xfer. To stop these, we propose that when the kernel is in this state any stoppable process executing a backward jump is transferred to the stopped queue. When all processes on the ready queue are unstoppable, control returns to the garbage collector. When the garbage collector has examined the frames to its satisfaction, it tells the kernel; the kernel

leaves this "stop stoppable processes" state and transfers the processes that are on the stopped queue onto the ready queue.

Note that when a process is stopped by the above algorithms, its evaluation stack is in a very simple state - the garbage collector never needs to know about the transient values placed on the evaluation stack, except those present at WAIT, during monitor entry, during an Xfer, and during a backward jump. (There may in fact be items on the evaluation stack during a backward jump - we may have to require that the compiler doesn't do this if the values are address-containing.)

While the garbage collector has stopped stoppable processes, it must take care not to invoke any monitor that is locked by a stopped process; it is believed that this can be arranged.

Inter-Office Memorandum

To	Cedar Interest	Date	May 7, 1980
From	Roy Levin	Location	Palo Alto
Subject	Guidelines for Signalling in Cedar	Organization	PARC/CSL

XEROX

Filed on: [Ivy]<CedarDocs>Style>SignallingGuidelines.bravo

This memo offers a methodology and some guidelines for use of the Mesa signalling mechanism by Cedar programs. In addition, it proposes some modest extensions to Mesa that make the methodology more convenient to apply in practice. The proposals of this memo are intended to supplement (and in some cases supersede) the guidelines adopted at the Cedar open meeting of March 27.

Acknowledgements

This memo has benefited from suggestions and comments by Andrew Birrell, Jim Horning, Butler Lampson, Jim Mitchell, Ed Satterthwaite, and Howard Sturgis.

Introduction - Philosophy and Assumptions

We want the Cedar programming environment to include a rich collection of *packages*. A package supplies some sort of useful *abstraction*, which is manipulated through a well-defined *interface*. The interface should specify all that is needed to use the package correctly; much of this specification is in a machine-interpretable form. If the abstraction is robustly implemented by the package, the interface is likely to contain some information about exceptional situations that may arise, some of which may imply partial or total failure of the abstraction. This simply recognizes that the real world is not made up of abstractions but of concrete entities that occasionally fail to support the abstractions they implement. The specification and handling of these failures fall within the province of an exception handling mechanism.

It would be convenient if the mechanisms in Cedar for specifying and handling exceptions directly supported this view of the programming environment. However, the existing exception reporting and handling mechanisms of Mesa are more primitive than those required by this model, and to redefine them to match the model would be a more ambitious undertaking than is appropriate at this point. Indeed, there are some research problems that would need to be solved first. Accordingly, no sweeping revision of the exception handling semantics is proposed. Instead, we propose to establish some conventions and guidelines for using the existing Mesa mechanism (or a slight variant of it) for exception handling in Cedar packages. The specific suggestions for language changes are intended to have only a modest effect on the compiler and supporting runtime system. If more serious changes to the language could be considered, the proposal here would be substantially altered.

The Mesa language defines a means of transferring control from a program that detects an exceptional situation to a dynamically-determined context or contexts where the situation may be handled. This control transfer is very similar to a procedure call, except that the binding to the callee is performed dynamically when the exception is raised. In fact, the callee(s) is(are) selected from the call stack of the process in which the exception is raised. Like normal procedures, these exception handling "procedures" can accept parameters and return values. Unlike normal procedures, they are set off by a distinctive syntax that clearly identifies them as code for the exceptional case. In addition, the cost of invoking these procedures is significantly greater than normal ones, in part because the implementation minimizes the normal case overhead incurred by the existence of such code.

Experience has shown that unconstrained use of the Mesa signalling mechanism complicates the construction of robust packages. In the past, two important principles have been overlooked or ignored:

- The client of a package must know what exceptional conditions it can raise and when.
- The exceptional conditions raised by a package must be intelligible to its client(s).

We should recognize that the exception handling structure of many existing Mesa programs and packages has been significantly influenced (warped) by the failure of lower level packages, including the standard Mesa-supplied ones, to observe these rules. In Cedar we have the opportunity to start over, unbiased by existing interfaces.

The next two sections of this document categorize signals and the ways in which they should be handled. The following section then proposes a methodology for using signals in the Cedar programming environment. The final section suggests some practical guidelines that support this exception handling methodology.

Classes of Signals

Mesa signals are used for a variety of purposes. To understand the motivation for the proposal below, we need an informal classification of these purposes. In the characterizations that follow, the term "signal" is used in a general way; in later sections we will distinguish between **SIGNAL**s and **ERROR**s.

Note: in the remainder of this document, we will only be considering signals that propagate *across abstraction boundaries.* We recognize that, within the implementation of a single abstraction, the Mesa signalling mechanism may be effectively employed in less stylized ways than are proposed below. Therefore, we do not impose requirements or restrictions on its use in such cases.

Internal Programming Errors

Situations that are expected never to arise, or which reflect some internal inconsistency in the data structures of a module or abstraction are often reported by raising a signal. For example, a loop that searches a list for some entry that "must" be there may (should) raise such a signal on its **FINISHED** exit. Some existing systems and packages use the unnamed signal **ERROR** for this purpose.

Caller Programming Errors

A robust (suspicious) program validates its inputs before attempting to perform a requested function. Illegal or inappropriate input parameters (e.g., an object handle which, upon inspection, appears not to point to a valid object) are often reported to the caller by a signal. If Mesa procedures carried formal input/output assertions, these signals would be raised when the caller failed to satisfy an input assertion. Since input assertions may specify conditions involving the state of the abstraction, as well as the parameters to a specific operation, these signals may report errors other than supplying bad parameters, e.g., calling procedures in an incorrect sequence.

Abstraction Failures

Serious abstraction failures other than internal programming errors are often reported by signals. In a formally specified system, these signals report the inability of an abstraction to satisfy a particular output assertion. Whether a particular problem is an abstraction failure or a caller error is sometimes difficult to decide and depends upon the specificity of the abstraction definition. For example, a symbol table abstraction may have an upper bound on the number of <name, value> pairs it can maintain; a call that causes the limit to be exceeded might reasonably be interpreted as either a caller error or as an abstraction failure. In principle, a formal specification would include the size limitation in the input assertions for the relevant operations and thus would report an overflow as a caller error. In practice, however, such limitations are not always advertised and thus a symbol table overflow might more reasonably be viewed as a failure of the abstraction. Furthermore, it is not always practical to express these limitations as input restrictions, particularly if the limitations can change dynamically (e.g., as a result of resource constraints -- see the next paragraph).

As a rule, we strive for simple abstractions and therefore occasionally specify the normal-case behavior of certain abstractions in an unrealistically simple way. Typical examples are basic resources: primary memory, disk memory, communications channels. Our normal-case specification is likely to treat these resources as offering infinite, error-free capacity, yet we know that we must occasionally expect errors or an inadequate supply of resources. We include signals in the specification of the abstraction to report precisely those situations that force the implementation to violate the simple, normal case behavior.

Informational Signals

The preceding signal classes all address errors or failures of some kind. However, it is occasionally useful to be able to report information to higher levels by means of a signal. Such a signal typically carries the connotation of "unusual event" rather than "error". For example, a procedure that performs a disk transfer may raise an informational signal if it encounters repeated transient errors, giving the caller the opportunity to relocate the data elsewhere on the disk. In a given situation, the choice between an informational signal and a return code is generally based on the frequency of occurrence and severity of the condition, aesthetics, and the normal-case return values of the procedure in question.

Handling Signals

We have characterized the situations in which signals of various classes are raised. Let us now try to characterize how these signals should be handled. that is, where the catch phrases should appear. To do so, we will consider our programming system to be structured into abstraction layers. At a given instant, the call stack of a process will contain procedure activations belonging to several layers, and if we have a global understanding of the system, we can identify "runs" (of successive activations) that all belong to the same layer. We will be concerned with the behavior of signals of the various types as they cross the abstraction boundaries that separate the system layers.

Throughout the remainder of this section, we use S to refer to an abstraction layer that raises a signal, C to refer to the client layer that uses S, and X to refer to the client layer that uses C.

Internal Programming Errors

When S detects an internal programming error, the abstraction and its implementation are in an unknown state. Any attempt to make progress using the abstraction is fraught with peril, since monitors may be locked, data structures may be inconsistent, etc. By definition, the implementation of the abstraction is not prepared for such a situation to arise, and therefore has no business catching such a signal. Let us suppose, for the moment, that C is trying to be robust and respond gracefully to failures of the packages it uses. Although C has no way to interpret the specific meaning of the signal (since it relates to the internal state of S), C would like to catch it and attempt to restore his own abstraction to a well-defined state without making further use of S. C may not be able to mask S's malfunction completely and may be forced to signal an abstraction failure to his client, X. However, once caught by C, the signal from S will not propagate any further.

There is a slight difficulty here. The specific nature of the signal from S is unimportant (and meaningless) to C; all that matters is that it represents a lower-level programming error. It is therefore pointless to require the specification of S's abstraction to include the specific signal name. Furthermore, future development of S should permit the inclusion of additional signals that report internal programming errors without forcing them to be included in S's interface specification. C needs a catch phrase that says, in effect, "catch any programming error, but I don't care specifically what it is". This is a restricted form of ANY, which presently exists in Mesa. We will propose a specific mechanism below for this purpose; for now, assume that such a facility exists. Thus, signals used to report internal programming errors do not (and should not) appear in an interface specification.

C may not want to catch signals that report programming errors inside S. After all, figuring out how to back out gracefully of such situations is often difficult, and not every package (particularly early in its life) cares to address the problem. If C leaves S's signal uncaught, it will propagate upwards in the call stack and eventually cross the next abstraction boundary into X. The signal will be unintelligible to this higher-level abstraction, so it has no reason to catch it. However, from X's point of view, C has committed a programming error by allowing a low-level signal to escape. If X is trying to be robust, it will want to catch this signal and recover from the perceived failure of C.

Since we are treating internal programming errors as unrecoverable, signals reporting them should properly be declared as **ERRORs**.

Caller Programming Errors

When S discovers a violation of its input assertions, it raises a signal indicating a programming error in C. S's abstraction is still healthy and usable, but C is now in an awkward state, since it presumably made the erroneous invocation of S because of some undetected inconsistency in its internal structures. Although one might argue that C might wish to catch this signal and do something intelligent, in practice this is rarely possible. Instead, a programming error within C has been detected and the state of C's abstraction is now suspect. Thus, X will want to see a signal that it can recognize as a programming error within C (if X chooses to catch the signal at all).

We should note that, in principle, the signal raised by S is intelligible to C, since it reports a violation of S's specified input requirements. More importantly, however, the signal is intelligible to C's *implementor*, who will ultimately have to locate and repair the programming error inside C. Therefore, while in principle the semantics of the signal could be carefully defined in S's interface

specification, it is more important that the signal be understandable to the human looking at it from the debugger. In practice it is probably unnecessary to include such signals in the interface specification.

Typically, the state of S's abstraction is unaffected by the detection of a programming error in its caller. Thus, there is no need for control to return to S after the signal is handled. Accordingly, signals that report violation of input assertions should be declared as **ERRORs**.

Abstraction Failures

An abstraction failure in S that is reported by a signal to C *must* appear in the interface specification for S. As the client of S, C is the only abstraction layer in which the signal is intelligible, and only C can catch it. C may be able to recover from the failure of the lower-level abstraction, or it may be forced to raise its own abstraction failure signal. However, unlike the programming error case, C is entitled to assume that S is not hopelessly confused; even though the normal case action could not be performed, S's abstraction is in a well-defined state (albeit an unusual one) and may still be used by C.

If, however, C fails to catch the signal from S and allows it to propagate upward across the next abstraction boundary to X, X will want to treat the signal as a programming error in C. This is similar to the programming error cases discussed above, and the proposals below offer a mechanism for handling them in a uniform way.

Signals that report abstraction failures must leave the abstraction in a well-defined state that is intelligible to its client. Existing Mesa programs frequently do not clean up the abstraction state at the time that an abstraction failure is *signalled*, but wait until the signal is *unwound*. This is a risky business, particularly if the catch phrase attempts to perform any operations on the abstraction. (Such operations, being in the catch phrase, occur before the signal is unwound.) We adopt the view that S should restore its invariant *before* raising an abstraction failure signal, which ensures that C's catch phrase sees a coherent state of S's abstraction. Under this assumption, abstraction failure signals should be declared as **ERRORs**, since resuming them cannot be meaningful.

Informational Signals

Since an informational signal is raised by S primarily for the convenience of C, it obviously must appear in S's specification and interface. This signal, like an abstraction failure, can only be intelligible to C. Thus, if it were to propagate across the abstraction boundary to X, X would want to treat it as a programming error (again, the proposals below give details). Occasionally, S is asking C for advice on how to proceed in the exceptional circumstance it has detected. In such cases, the signal will have one or more return values that are used to communicate C's desires to S. Obviously, S's abstraction should be in a well-defined state when the signal is raised.

It is evident, then, that informational signals should be declared as **SIGNALs** and should generally be **RESUMEd**. However, on occasion C may want to **UNWIND** an informational signal (e.g., the information conveyed by the signal may be sufficiently distressing that C wants to abort its computation). Alternatively, S may (for C's benefit) define a particular return value meaning "please clean up and go away". We do not require that informational signals be **RESUMEd**, but we expect that this will be the normal case.

A Methodology for Using Signals in Cedar

With the background presented above, we can now propose a specific approach to exception handling in Cedar. The proposal has two parts: some methodological principles and some guidelines that support them. The specifics of the methodology are directly derived from the general discussion above. Recall that this methodology is only intended to encompass exception handling *across abstraction boundaries.*

Embodied in this proposal are some small language changes. Note, however, that much of the approach recommended here cannot be enforced by the compiler and runtime system without more substantial language modifications.

1) *An interface specification to an abstraction is defined by a* **DEFINITIONS** *module introduced by the following notation:*

    AbstractionName: ABSTRACTION = . . .

    The procedures defined in such an interface are said to lie "on the abstraction boundary".

    Motivation: Since the meaning of signals changes when they propagate across an abstraction boundary, it is necessary to be able to recognize such boundaries at runtime. When the compiler is compiling a program module and comes across a procedure which is exported to a definitions module labelled as an abstraction, it will mark the procedure in such a way that the runtime signalling mechanism can recognize activations of that procedure as being on the abstraction boundary. Implementation details available on request.

2) *Every signal that crosses an abstraction boundary identifies an exceptional condition in precisely one of the four categories described above.*

3) *Internal programming errors and caller programming errors do not appear in the interface specification of the abstraction that raises them. Informational signals and signals reporting abstraction failures must appear in the interface specification of the abstraction that raises them.*

4) *Signals that appear in an abstraction definition should be accompanied by a precise description of the state of the abstraction when they are raised and the procedures in the interface that can raise them. If unwinding an informational signal alters the abstraction state, that change should be described as well.*

5) *Abstraction failure signals must be declared as* **ERROR**s. *Informational signals must be declared as* **SIGNAL**s.

6) *Signals used to report internal programming errors must be declared using the notation:*

    ErrorName: PROGRAMMING ERROR;

    Signals used to report caller programming errors must be declared using the notation:

    ErrorName: CALLING ERROR;

    Note 1: Both **PROGRAMMING** and **CALLING ERROR**s may be public or private, variable or constant, and may have parameters, just as present-day **ERROR**s do. The purpose of the additional adjective is to distinguish the error so that the runtime signalling mechanism can implement the semantics discussed above. We will describe the way in which it does so shortly.

Note 2: In present-day Mesa, the unnamed signal **ERROR** is a debugging nuisance if, as is often the case, complete symbols and source files are not available. Without this information, it is difficult for the implementor to determine the precise condition that caused **ERROR** to be raised. In Cedar, we expect that full symbols and source files will always be available to the debugger; hence, an unnamed signal **PROGRAMMING ERROR** becomes useful and even desirable.

7) *The Cedar runtime signalling mechanism operates in the same way as the present Mesa mechanism with the following differences:*

a) *As the signal propagates upwards through the call hierarchy, the signalling machinery watches for abstraction boundaries. When the first abstraction boundary is crossed, the signaller notes this fact but takes no other action.*

b) *When the second abstraction boundary is crossed, a signal has, in effect, escaped from the only abstraction in which it could reasonably be caught. The signaller ceases propagation of the signal and implicitly raises*

UncaughtSignal: PROGRAMMING ERROR;

It propagates this signal beginning at the point where propagation of the original signal ceased. Note that this is not the same as the current treatment of nested signals (see Mesa language manual section 8.3).

c) *When the signaller performs the transformation described in b), it does so without losing the original signal, permitting the signal to be examined in the debugger (if* **UncaughtSignal** *goes that far).*

Note: This mechanism is appropriate for strictly layered abstractions. It does not work well in the situation where multiple layers in the call hierarchy correspond to the same abstraction. This can occur with so-called "call-back" procedures, of which enumeration procedures are perhaps the most familiar example. A signal raised by the call-back procedure which is intended to be caught by the invoker of the enumeration procedure should really be considered to cross no abstraction boundaries rather than two. Perhaps we need to qualify such signals when they are declared, to prevent their implicit conversion to **UncaughtSignal**. Something like

ErrorName: LOCAL ERROR;

asserting that the error handling will be local to the abstraction that declares this signal, could be used to invoke this special treatment. Alternatively, we might suppress the implicit conversion to **UncaughtSignal** if the signaller and handler are in the same module. Ultimately, we would like to have a tighter association of abstraction and module anyway.

8) *Abstraction failure and informational signals are caught in the usual way. Internal and caller programming errors cannot be caught by name, since they do not appear in the abstraction's interface specification. Signals in these classes may only be caught using the following syntax:*

CatchItem ::= Catch | PROGRAMMING => Statement |
CALLING => Statement

Motivation: As we observed earlier, there is no point in trying to catch internal and caller programming errors by name. A "generic" catch is primarily needed when a package is trying to be robust in the face of failures by lower-level packages. Since uncaught signals escaping from lower levels show up as a **PROGRAMMING** error (see item 7), they will be caught by this generic construct.

Note 1: This definition is intended to replace the one in the Mesa language manual. effectively eliminating ANY as a means of catching signals that cross abstraction boundaries.

Note 2: In view of the semantics of CALLING errors, it is unlikely that a catch phrase for such errors would do anything except raise a PROGRAMMING error. While this is perfectly acceptable and appropriate, the signalling mechanism will implicitly map uncaught CALLING errors to a PROGRAMMING error (specifically, **UncaughtSignal**). Thus, the absence of such a catch phrase is consistent with the methodology (and can save the programmer considerable coding).

9) *When an abstraction failure or informational signal is raised, the signalling abstraction must be restored to a coherent state before the signal is raised, not when it is unwound.*

The motivation for this requirement has already been discussed. Although this requirement imposes some inconvenience upon the implementor, since (s)he must arrange to clean up data structures, restore invariants, and unlock monitors before signalling, it ensures that the client is never executing when the abstraction is in an inconsistent state.

Guidelines in Support of the Exception Handling Methodology

This section suggests techniques which support the methodology presented above and avoid various pitfalls in the use of the Mesa signalling facilities.

1) *Catch all signals defined in the interfaces to the abstractions you use. If you cannot proceed intelligently, raise a signal of your own (which should be defined in your abstraction's interface).*

2) *Don't bother to catch* PROGRAMMING *or* CALLING *errors unless you are trying to build a bullet-proof package.*

3) *When you raise an informational signal, be prepared for an* UNWIND.

4) *Try to keep* ENABLE *scopes as small as possible. You will minimize confusion about which procedures raise which signals.*

You'll also get more compact code, given the current implementation of ENABLE.

5) *Use* RETURN WITH ERROR *only when there is a good reason for doing so.*

RETURN WITH ERROR is only appropriate in an ENTRY procedure when you want to unlock a monitor before raising a signal. In other circumstances, RETURN WITH ERROR is unnecessary and is likely to confuse anyone reading the code. It may also complicate debugging by releasing a potentially interesting local frame. If you think otherwise, you are probably confused about the semantics of this construct. Reread sections 8.2.3 and 10.5.2 of the Mesa language manual.

6) *When defining internal or caller programming error signals, don't include parameters unless they aid in debugging. Also, ask yourself why you are bothering to name your* PROGRAMMING ERROR *signals.*

Recall that these signals are only caught by "generic" catch phrases, hence their names and parameters are not available to the program that catches them. Don't bother to include as parameters values that you will easily be able to display with the debugger. Remember that, in Cedar, you should always be able to display things in complete symbolic form, i.e., you won't have to put up with missing source files or compressed symbol tables.

7) Beware of undesirable interactions of UNWIND *and* INLINE *procedures.*

It is common to insert INLINE adjectives after a program is working to improve its performance. This can lead to surprises in certain exception handling situations. For example, if, in the following code,

```
P: PROCEDURE =
 BEGIN
 . . .
 Q[ . . . ! Trouble => GO TO Cleanup];
 . . .
 EXITS
  Cleanup => . . . ;
 END;

Q: ENTRY PROCEDURE =
 BEGIN
 ENABLE UNWIND => NULL;
 . . .
 ERROR Trouble;
 END;
```

procedure Q is made INLINE, its UNWIND catch will not be executed and consequently its monitor will not be unlocked. Although this is certainly a bug, it is awkward to fix and will probably be with us for some time. The problem here is that P and Q have the same frame at runtime. UNWINDs are defined to propagate up to, but not including, the frame in which the UNWIND was triggered. Therefore Q's UNWIND catch phrase will be executed only if it is an out-of-line procedure.

Some Notes on UNWIND

The proposed methodology substantially reduces the need for implementors to write UNWIND catch phrases. UNWIND catches are intended to permit last-minute cleanup of an abstraction before control is regained by its client. However, except in the case of informational signals, a program constructed in accordance with the above methodology will have no cleanup to perform. Indeed, this will often be true for informational signals as well.

Of course, this simplification is not free. The cleanup must now occur before the signal is raised, which implies that the signalling abstraction cannot simply raise the signal at the point of detection. Rather, it must first "back out" of its computation, possibly using private signals to get control to the "top level" of its implementation, unlocking monitors and cleaning up internal structures along the way. The top level would then raise the signal defined in the abstraction's interface.

One could reasonably argue that, were it not for the desire to see the call stack when an uncaught signal reaches the debugger, we could adopt a termination model (as in BLISS or CLU) for all signals except informational ones (i.e., the call stack is pruned during signal propagation, and there is no concept of unwinding). Indeed, the methodology corresponds closely to such a model; if the implementation never invoked UNWIND catches for signals other than informational ones, we would have similar semantics.

In this context, it is perhaps worth mentioning an independent proposal for redefining the semantics of an **UNWIND** catch phrase. It has been suggested that the body of an **UNWIND** catch always be executed just before the local frame of the enclosing procedure is released, whether due to **UNWIND** or **RETURN**. This would replace the all-too-familiar:

```
P: PROCEDURE =
 BEGIN
 p: NodePtr ← AllocateNode[...];
 BEGIN ENABLE UNWIND => FreeNode[p];
 ...
 Q[args];
 ...
 END;
 FreeNode[p];
 END;
```

with:

```
P: PROCEDURE =
 BEGIN
 p: NodePtr ← AllocateNode[...];
 BEGIN ENABLE UNWIND => FreeNode[p];
 ...
 Q[args];
 ...
 END;
 END;
```

Under our proposed methodology, these situations would be less common, since **UNWIND** would primarily appear within abstractions. Still, this proposal is appealing, for it permits cleanup code to be written in only one place.

It has also been suggested that the debugger could permit unwinding on request. This could be useful when a missing catch phrase has caused an uncaught signal. The reader is invited to consider the ramifications of such a facility.

Inter-Office Memorandum

To	Cedar filing, Cedar language	Date	May 13, 1980
From	B. W. Lampson	Location	CSL, Palo Alto
Subject	System modelling, May 1980	File	[Ivy]<Lampson>ModelsNow.press

XEROX

Last edited by BWL, May 13, 1980, 2:14 PM

This memo contains a proposal for a system modelling facility which is consistent with the long-range goals of Cedar, but can reasonably be implemented now and used with the current Mesa system. Eric Schmidt has completed large parts of an implementation. Comments are solicited on two questions:

> Is it right? I.e., is it consistent with the long-range goals of Cedar? Does it address all the important issues, and address them correctly?
>
> Is it feasible now? Can this proposal, or the subset of it indicated in the comments on current implementation which appear here and there, be implemented in time to be useful for Cedar development?

Note: this is definitely not a user's manual.

Acknowledgements

This approach to system description originated in the design of the Mesa binder, which I did with Charles Simonyi, John Wick and Rich Johnsson. I am indebted to the members of an SDD working group on consistent compilation for many fruitful discussions. Jim Horning and Andrew Birrell also helped to shape these ideas during meetings of the Cedar filing design group. Eric Schmidt has pointed out a number of problems and focussed attention on making the modeller useful.

1. Framework

A *system model* is a piece of source text which contains a complete description of a Cedar system; i.e., all the information needed to construct the system is present. A model may have (formal) *parameters*, however, and in that case (actual) *arguments* must be supplied to completely define a system.

There is code to support system models (called the modeller). It:

> Invokes the compiler, binder etc. to construct the system described by a particular model, and tries to keep track of when this work has already been done and need not be repeated;
>
> Provides convenient ways of updating a model to reflect new versions of the component modules or models (would you believe *modelules*, or *modulels*) which come along during the development of a system;
>
> Supplies the necessary interfaces to the file systems which actually store the information, copies data to local disks when needed, and so forth.

Remark: One of the reasons this has been hard to get straight is that it requires understanding polymorphism (type parameters) to some extent, because Mesa actually has polymorphism. It only shows up in definitions modules. was not recognized very clearly, and is not done right. Some of the problems came out more clearly as a result of the desire for exported types, which are another form of polymorphism. In fact, global frames are also exported types. Furthermore, any PRIVATE type probably should be exported, though the current language provides no protection against mixing up two implementations of such a type. This didn't seem important because it doesn't affect the integrity of the built-in types. ∎

Remark: Another persistent source of confusion is that details of the implementation have often been taken as conceptually fundamental. In this exposition, such details are relegated to their proper place (I hope). ∎

The description in this memo is couched in terms of Mesa. However, the entire scheme is applicable to any collection of source files and programs for converting them into a running system (*translators*), provided only that *all* the parameterization can be specified when the translators are invoked (most translators currently do not have this property, usually for non-essential reasons). I expect that in the long run the modeler will be usable for other such applications, e.g., font shuffling and hardware design automation, with only minor changes to the existing translators for those applications.

2. Units

A particular system model with no formal parameters contains (in principle) all the source text from which the system is generated; i.e., it is like a card deck for the entire system. In fact, of course, this text is stored on a number of files scattered throughout the file system. To take account of this, we introduce the notion of a *unit*, which is a piece of source text with no free identifiers (but perhaps with formal parameters). Because it has no free identifiers, a unit can stand alone, and need not be part of a larger structure in order to be meaningful. A unit is *immutable*; its contents never change. Of course, it is possible to use the contents of a unit as a basis for constructing a new unit; this process is often called *editing*. A model itself is a unit, and so is a Mesa module. Smaller units, such as procedures or even statements, are also possible (but not in the initial implementation).

The immutability of units is an essential aspect of the scheme, since it allows a name to stand unambiguously for the contents of a unit, and ensures that no property of the named object can change (an edited object would have a new name). Among other things, this means that information about the object which is cached, e.g., in a data base maintained by the modeler, cannot become invalid and hence never needs to be checked. The price for this, of course, is lots of new names, and hence it is essential to have convenient ways of updating a model to refer to a new object; see section 6 below.

Each unit has an *identifier*, which is unique throughout the entire universe of discourse. The identifier is a string, and is composed of *parts* which are alphanumeric strings, separated by dots or bangs; e.g., Compiler.Parser.RepairError!16. Uniqueness is guaranteed by a decentralized registration procedure. A *naming authority* for the identifier *i* is denoted *i*+ or *i*!*d*↑, where *d* is a string of digits called a *version number.* An identifier *j descends* from *i* if

i=*p*+ and *p* is a proper prefix of *j*, or

i=*p*!*d*↑ and *p*!*e* is a prefix of *j* for any *e*>*d*.

Two identifiers *conflict* if they are equal, or either descends from the other. An authority contains a *log* which is a list of the identifiers it has issued; it may issue any identifier which descends from it and does not conflict with a identifier already in the log.

In the initial implementation units correspond to files. Since our file systems cannot ensure the immutability of a unit, or any fixed correspondence between name and contents, it seems desirable for the modeller to do this. We therefore append to each file its identifier (to fix the correspondence) and a checksum of the file contents including the identifier (so that immutability can be checked), in the form <<<identifier: checksum>>> *(or something like that).* If this is done by

looking for <<<...>>> near the end of a file and replacing the ..., the client can embed this construct in a comment for all languages and file formats we currently have (except pictures). The intention is that the file system name of a unit will be the same as its identifier, possibly with a prefixed directory.

Note that there is nothing wrong with having multiple copies of a unit in the file system, since all are identical and immutable (of course multiple copies of a naming authority are bad). In fact, we *expect* multiple copies to occur, when units are copied to local disks.

Obviously it is necessary to have some way of knowing where to look in the file system for a particular unit, since exhaustive search is impractical. This could be done with a special data base, but it is easier to include this information in the system model, as described below, and we will start that way. Note that such file system pointers are simply hints; if they are wrong, the modeller may not be able to proceed, but it will never use the wrong unit. Presumably it could also ask the user where to look in case of difficulty.

I don't yet have a position on whether we should implement naming authorities now, or do that manually for the moment. I think that on Juniper it wouldn't be a big deal.

3. Modelling language

The following grammar describes the modeling language. It is taken from the simplest abstract grammar for Cedar/Mesa in the current draft of the Cedar abstract machine description. Hence, if no mistakes have been made, it is a subset of (abstract) Cedar/Mesa, except for the stuff in bold, which was added specifically for modelling. Details are open to negotiation; in particular, (in)consistency with the concrete syntax has not been carefully considered.

Notation for the grammar:
Parentheses are for grouping. Operator precedence is | (lowest), ?, }.
Item | item means choose one.
?item means zero or one items.
{item}sep means one or more items separated by sep (which may be empty).
Terminals are punctuation other than (){}?| and --, or quoted, or in SMALL CAPS.
Subscripts, and from -- to end of rule are comments, not part of the grammar.

exp ::= {part}; | {part},

part ::= ?id : call | call | ***: id | id: * | *id** | LET **group ?(== call)** | OPEN **call**

id: call declares id; the call specifies type (and perhaps value). All other cases are short for various uses of this.
If id is omitted (: call). the call must specify a value as a unitId, and its principal part is taken as the missing id.
If id: is omitted, the part is said to be *unnamed.*
*: id is short for idImp: id; id: * is short for id: TYPE, idImp: id; *id is short for id, idImpl.
LET group declares each id in the group; the names in the call are ignored, and it is bound to the group by type equality. This is short for individual declaration of the ids, with the proper part of the call, selected with ., for the value of each.
OPEN call is equivalent to LET group==call, where group is the type of the call (which must be a group).

call ::= primary | primary group | primary == call |
primary PLUS primary | primary THEN primary |
SELECT call FROM [{$call_1$ => exp_2};] -- exp_2 whose $call_1$=call

proc[] binds the group to the proc params, evaluates the proc body, and binds this value to the proc result; see § 4.
== only when the primary is a type and the call a compatible value

primary ::= literal | id | **unitId** | "(" call ")" | primary . id | group
TYPE **?id** | STRING | PROC group ?(RETURNS group)

group ::= [?exp]

unitId ::= @ location {?* uiPart}. ?(! {digit})

* marks *principal* part; if it is absent, the last part is principal.

location ::= ?([uiPart])?{uiPart}>
uiPart ::= id | id ↑

Operators are left-associative.

We may need integers and enumerations here at some point, e.g., to state compiler parameters?

A type is one of the primitive types (currently only STRING),an interface type TYPE id, or a PROC type, or an exp type. PLUS and THEN are built-in operators on two objects of the same interface type.

An expression is a list of parts, each with an optional name, a type, and an optional value:
name: *type* = = *value*
Since there is no assignment in this language, the value must be supplied on the spot, except when the exp defines a parameter or result type. If the value is present, the type may be inferred from it:
name: *value*
For historical reasons, either commas or semicolons may separate the parts. The identifiers of the exp denote the corresponding calls throughout the exp, including any nested exps; redeclaration is not permitted. Calls are used to define and apply procedures, as explained in § 4. Various common cases can be abbreviated, as described in the definition of part above. Note that omitting the id: for two parts of an exp with the same type is useless, since the resulting exp cannot be bound to anything.

One expression *v* may be *bound* to another *e* if, for each part of *e*, there is a part of *v* with the same name, or a unique unnamed part of *v* with the same type. For each part of *e* with name *n*, the result has a part
n: *e.n.*Type = = *v.n.*Value;
*v.n.*Type must conform to *e.n.*Type.

The unitId stands for the text of that unit. The part of it following the location (with the optional * removed) is the unique name of the unit, and the location is a hint about where to find it. In a unitId, a naked identifier *x* denotes the STRING literal "*x*", and *x*↑ denotes the value of the identifier *x*, which must be a STRING (and is probably a parameter). The only alternative I can find for this somewhat peculiar syntax is to require the literals to be quoted explicitly. That seems very clumsy, since a literal is the normal case inside a unitId. A * marks the *principal* part of the unitId, which indicates the identifier to be declared. E.g.,
:@[Ivy]<Mesa>4.3>Mesa4.*Parser.CedarExtensions!13
declares Parser to have the value which is the text of that unit. The * is *not* part of the unit name. If no part is starred, the last uiPart is the principal part..

The id following TYPE allows a crude check that the type (defs module) being obtained is the one expected: a value of type TYPE *X* is produced by a module which starts *X*: DEFINITIONS. The basic job of the type system is to ensure that an argument has the expected properties; this is the one place where that job cannot be done properly, since it is precisely the statement of those properties which is being supplied. The identifier serves as an informal abbreviation of the missing information. If it is omitted in a part (i.e., [... *X*: TYPE ...]), the part's id (i.e., *X*) is used instead.

We write *A*→*R* in this document as short for PROC *A* RETURNS *R*, or for PROC [*A*] RETURNS [*R*] if *A* and *R* have only one part.

4. Parameters

In a call the primary must be a proc or a type. In the former case the meaning is to bind the arguments in the call to the parameters of the proc, and then evaluate the body of the proc. The types of the arguments must match the types of the parameters. This is ordinary procedure call. Note that the modeling language is purely applicative; it does not have assignment. Hence parameter binding is just substitution of the arguments for the parameters in the body. The use of explicit NEW and UNNEW operations, and other modification of interfaces at runtime, makes the computation which produces the running system non-applicative. This is not necessarily bad, but there is a price in loss of clarity. If the modelling language is also to be used for describing the state of the running system, there will have to be a way to prettyprint that state as a model. This is an issue for the debugger.

The result of applying a type is to coerce the argument exp to have that type; thus CARDINAL[3] or CARDINAL==3 is a value of type CARDINAL. In the modelling language, this is only interesting when the type is a proc type.

The result of applying a proc type, $(A \rightarrow R)[i_1: e_1, ..., i_n: e_n]$, is a procedure whose body is the argument expression, with parameters which are the parts of A. For each part $i:T$ of R, there must be a part $i:e_i$ of the body, and e_i must have type T. The value of the call is $(i_1:e_{i_1}, . . ., i_k:e_{i_k})$; unmatched parts of the body are discarded. For example, consider

PROC [*a*: ..., *b*: ...] RETURNS [*x*: ..., *y*: ...]==(*v*: ..., *w*: ..., *x*: ..., *y*: ...)

The body declares *v*, *w*, *x* and *y*, and may refer to all these and to *a* and *b*. The result is the values of *x* and *y*. Thus $\lambda x.e$ in the untyped lambda calculus would be written $(x:T \rightarrow y:U)[y:e]$ in this notation, for some suitable T and U.

A program module has a proc type, with parameters specified in the DIRECTORY (type parameters) and IMPORTS (interface record parameters) clauses, and results specified in the EXPORTS clause. The imported and exported interfaces have types which are the values of defs modules; hence their values are just records of procedures and (pointers to) variables, plus constant components which are not stored. With exported types, this is no longer true, since an exported type is a part which can vary from one instance to another; however, it still isn't necessary to store the typeid, though it may be convenient to do so. A definitions module with no DIRECTORY has type TYPE *X* (or []→[*X*: TYPE *X*]); if it has a DIRECTORY its type is [...]→[*X*: TYPE *X*]. Thus

```
DIRECTORY A: FROM "Afile", B: FROM "Bfile";
X: DEFINITIONS = { ... };
```

has the type [*A*: TYPE *A*, *B*: TYPE *B*]→[*X*: TYPE *X*]; we could have elided the tags and written this [*A*: TYPE, *B*: TYPE]→[*X*: TYPE]. Note that "Afile" and "Bfile" do not appear; they are the *arguments* for the parameters *A* and *B* of *X*, and as such do not belong in the definition of *X*.

As explained above, the *X* in TYPE *X* is not really necessary. Without it. however, a TYPE parameter would accept any DEFS as an argument, and this could lead to considerable confusion if the wrong module was supplied by mistake. If *X* is specified, then it must match the *X* of the argument; this check is likely to catch most such errors.

Of course models can be parameterized also. In fact, normally a model will be a call of the form

(PROC group RETURNS group) [...].

There is no restriction on the type of a parameter. In particular, a parameter may have type TYPE; a type parameter may then appear in the result type of the proc type. This is not a frill; otherwise it would not be possible to treat a module as a unit with only parameters and no free identifiers. For example, suppose we have a program module *Ximp* which exports (implements) a definitions module

Xdefs=*Xtag*: DEFINITIONS={...}.

The model for this is:

X: TYPE *Xtag* = @*Xdefs*!14;
xi: *X* = @*Ximp*!7[*X*];

Here @*Ximp*!7 has the type *X*:TYPE *Xtag*→*xi*:*X*. From now on we will use *X* for *Xtag*, and elide it where possible. Thus our model becomes

X: TYPE = @*Xdefs*!14;
xi: *X* = @*Ximp*!7[*X*];

and *Ximp* has type *X*:TYPE→*xi*:*X*. Note: most current Mesa users would have *Xdefs* where we have *X* here, but this is strictly a matter of convention; Pilot, for example, follows the opposite rule and has *X* where we have *Xdefs*. In practice the unitId will likely be the same (or have a part which is the same) as the type tag, but we keep them different to emphasize the different kinds of objects involved.

It is also possible to supply arguments for a subset of the parameters; the result is then a proc with the remaining parameters as its parameters, the same result type, and a body which is the body of the original proc with the supplied arguments substituted for the corresponding parameters. Thus

([*x*:*T*, *y*: *V*]→*U*)[...*y*...*y*...])[*y*: *e*] = (*x*:*T*→*U*)[...*e*...*e*...]

Such a partial call is called a *projection. There is an unfortunate conflict between this notation and the Mesa notation for defaulting arguments.*

Projections are interesting because the compiler produces them: a module cannot be compiled until all its type (defs) parameters are supplied (except the opaque (exported) types). Thus, if we extend the example above with another interface *Y* and a module which imports *X* and exports *Y*, we get

X: TYPE = @*Xdefs*!14;
Y: TYPE = @*Ydefs*!1;
xi: *X* = @*Ximp*!7[*X*];
yb: (*xi*:*X*→*yi*:*Y*) = @*Yimp*!3[*X*, *Y*];
yi: *Y* = *yb*[*xi*];

Here @*Yimp*!3 is a unit with the type [*X*:TYPE, *Y*:TYPE, *xi*:*X*]→[*yi*:*Y*], and *yb* corresponds to the BCD: it still needs a value of type *X* to produce a usable interface. Note: it is wrong to say that *yb* *is* the BCD; it is just a projection of *Yimp*. The model has no concept of BCD.

If we treated procedures, rather than modules, as the smallest units, we would see that a global frame is just like any other interface (observation due to Andrew Birrell). That is, its type must be known to the compiler, and the instance can be supplied later. Of course the implementation is significantly different at the moment.

The resemblance of this to C/Mesa is certainly not accidental. The difference is that in C/Mesa there is no explicit description of, or control over, the type parameterization: which argument is supplied for a type parameter is entirely determined by what happens to be on the disk under that file name.

5. Abbreviations

The example above shows that writing out all the details is rather wordy. Things are improved by the following coercion:

x[] ⇒ *x*[a_1, ..., a_n] whenever *x*: [a_1: T_1, ..., a_n: T_n]→*U*

This might be called "strong deproceduring." With it, we can omit all the argument lists in the example.

We also note that the type generally doesn't have to be written in a part, since it can be inferred from the value. If the value is a procedure, the ambiguity about how many parameters should be supplied is resolved by supplying all of them (*not* none of them, which might seem more natural, but is certainly less useful). Hence if a projection is required, its type must be stated explicitly. Fortunately, in many cases there is no reason to mention the projection explicitly, especially if it is applied only once.

With these abbreviations we can write the example more concisely:

X: @*Xdefs*!14;
Y: @*Ydefs*!1;
xi: @*Ximp*!7[];
yi: @*Yimp*!3[];

Both omissions of course reduce the value of the model for documentation. Also, the modeller will probably want to keep the types around, since otherwise it must examine all the files. This suggests that the text of the model with all the information should be the unit, and it should be possible to get it printed with less detail. It should also be possible tell the modeller to examine the files and update the type information. In fact, the types are exactly the information which the current include checker recovers from the BCDs. Using the BCDs for this purpose is contrary to the modelling philosophy, however, which is that only the sources are real. It also has the drawback that everything has to compile before the information is available. Since only the directory, imports and exports lists need to be examined, it is no big deal to get the types from the source.

Using principal parts, and standardizing on *X*, *Y*, *XImp* and *YImp* as names, we can further abbreviate this to

:@*X*!14; :@*Y*!1; :@*XImp*!7[]; :@*YImp*!3[];

6. Editing and noticing units

When a unit is changed, the result is a new unit with a different identifier. Typically the change is done as part of the development of some system, which is represented by a model. For the change to have any effect on the system, it is necessary to produce a new model which refers to the new unit; then the system described by this new model can be built. We call this *noticing* the unit (the term is borrowed from the Interlisp file package).

Notice: [*M*: Model; *old*: Unit; *new*: Unit]→*M'*: Model

searches *M* and, recursively, any models to which it refers, for occurrences of *old*; each such occurrence is replaced by *new*. Of course, when this is done in a model, the result is a new model, references to which are in turn replaced. The resulting model refers only to *new*. When a new model is made during this process, it has to have a new identifier. If the old identifier has the form *n*!*d*, the modeller obtains *n*!*d*↑ (the SDD Librarian calls this checkout), and uses *n*!(*d*+1) for the new one. Otherwise it asks the user what to do. *Doubtless other reasonable strategies are possible. A nice one might be to append the user's initials* !1, *and then to count up the version.*

Noticing should be done automatically when a file is edited, so that the single user developing a program by editing modules will hardly be aware of the model. To this end, the modeller should be told when a file is changed by the IME editor, and hopefully also by Bravo (perhaps using the measurement log). There should also be a way to ask it to check all the write dates of files mentioned in a model, in case the incremental process gets confused. It seems highly desirable to package the modeler into either IME or the compiler, in order to avoid an extra context switch. Furthermore, the modeler should keep the current model (and any submodels which it had to expand because there wasn't a single BCD) in memory as a graph, so that the file doesn't have to be read and parsed on every edit.

We should also have *Add* and *Remove* operations;

Add[*M*: Model; *new*: Unit; {*before, after*}; *old*: Unit]

inserts *new* before or after the (unique) occurrence of *old* in *M*. It would be nicer to do these operations by editing a picture of the model, but that is a longer-range feature. With *Notice*, these should take care of all ordinary changes to the model during straightforward program development by a single person.

If this kind of development proceeds independently along two paths, both starting with *M*, one often wants to merge the results back into a single new model.

Merge[*Base*: Model; *M*1: Model; *M*2: Model]→Model

traces out the three models depth-first. At each level:

If	*Add to result*
Base.p=*M*1.*p*=*M*2.*p*	*Base.p*
Base.p=*M*1/2.*p*≠*M*2/1.*p*	*M*2/1.*p*
Base.p=*M*1/2.*p*, no *M*2/1.*p*	leave *p* out
no *Base.p* or *M*1/2.*p*	*M*2/1.*p*
Base.p≠*M*1.*p*≠*M*2.*p*, all models	*Merge*[*Base.p*:, *M*1.*p*, *M*2.*p*]
ELSE	error, or ask what to do.

Of course, there is no guarantee that the resulting thing makes any sense, but it does seem to correspond to current practice. If we don't go in for checkout, it will also be necessary to choose the right name for a unit being newly inserted.

A nice feature of *Notice* and *Merge* is that they still seem to do the right thing even if the analysis is pushed below the level of models and modules, down to individual procedures or further, to compound statements (but I am not advocating this for now).

7. Compiling and binding

In order to actually build a system, the modeller must get everything compiled and bound. Thus, for each call of a unit, it must collect BCDs for all the type (defs) parameters, and values for all the parameters corresponding to compiler switches, and call the compiler to produce a BCD. Then it must construct a C/Mesa .config file for the model (perhaps for each sub-model) and call the binder.

Obviously, we don't want this done from scratch after every tiny change. Hence the modeller must keep some kind of data base which tells it what BCDs already exist. Logically this has a very simple form: a projection $U[U_1, ..., U_1]$ always produces the same BCD (this is why all compiler switches, including the compiler version, which is not strictly a switch, must be included among the parameters; the deproceduring coercion will keep us from having to write these things). Hence the data base simply lists pairs <projection, BCD file>.

Questions to be resolved:

How to represent the data base. Eventually, use the Cedar facilities. In the short term if they aren't ready, perhaps keep a file with name derived from the unit name which lists all the BCDs and associated parameters. Or this file could lump stuff for a bunch of units, as long as it doesn't get too big. Or whatever.

When should old BCDs be removed? This is strictly a space-time tradeoff, since they can always be rebuilt from the sources.

Do we want to make any effort to make the grain of dependency finer, as Desoto did? Probably not.

We consider the details of how modules get loaded and started to be outside the province of the modeller. They are part of the runtime system.

CSL Notebook Entry

To	CSL Cedar Interest Group	Date	Revised January 13, 1981
From	Cedar database committee (Brown, Cattell, Suzuki)	Location	PARC/CSL, Palo Alto
Subject	Cedar database concepts and facilities (phase one implementation)	Files	[Ivy]<CedarDocs>Database> C+F*.memo, CFFig1.press, C+F.press

XEROX

Archive ID: yyCSL-xxx

Attributes: technical, Cedar, Database

References: [Ivy]<CedarDocs>Database>Consol.press (August 29, 1979)

Abstract: The Cedar database system is intended to provide a convenient, high-performance data management facility for use by all Cedar clients. The goal is an integrated, unified representation of data throughout a network, aiding information exchange and avoiding repeated implementations of specialized data management facilities. We present the basic concepts and capabilities of the Cedar database system phase one implementation, and specify the programmer's interface to the system.

TABLE OF CONTENTS

1. Introduction 2

2. Database concepts 3
   - 2.1 Three basic properties of databases 3
   - 2.2 Consequences of the properties 5
   - 2.3 Database versus persistent virtual memory 6
   - 2.4 Cedar database (phase one) aspirations 6
   - 2.5 Cedar database applications 7

3. System design overview 8

4. Client interface 11
   - 4.1 Types 11
   - 4.2 System tuplesets 12
   - 4.3 Procedures 14

5. Database design 19

6. A sample program 23

7. References 26

1. Introduction

Why a Cedar database system?

CSL needs a database system. Such a system will include many data management facilities that are currently re-invented by program after program, and thereby eliminate a large duplication of effort. It will also provide a capability that is needed by many programs but actually implemented by very few: a means of sharing information among programs that are distributed around the network. The lack of a database system has hampered research in this laboratory, with its special interests in the organization and communication of information.

Why this paper?

We hope that this paper will be read by potential users of the Cedar database system. It is certainly required reading for those who want to make use of the system we are building now.

This paper begins by developing database concepts that are necessary to put the Cedar database facilities in perspective. We do not expect most people in CSL to become familiar with database jargon, or to read a Mesa interface, in order to discover whether or not the database system can be useful to them. Thus we begin (Section 2) by defining the scope of the general database problem, and measuring our ambitions relative to this standard. We also describe some sample applications that might take advantage of our system.

The main body of this paper describes the interface to the Cedar database system. We first give an overview (Section 3), then present the full story (Section 4). This requires defining the class of data structures that can be stored in a database, and then describing the procedures for manipulating these data structures. For illustration, we then give an example of how a particular real-world information structure might be represented as a database, and discuss design tradeoffs (Section 5). Finally, we present a short Mesa program that manipulates a database (Section 6).

Building a database system is a substantial undertaking, and we are reluctant to specify the precise form of the eventual system without some experience with CSL applications. This paper describes an interface that we shall present to our first clients, starting in mid 1980. We plan to build other interfaces on top of the one described here, and eventually to integrate some set of database facilities into the Cedar Mesa language. Earlier documents (particularly [Ivy]<CedarDocs>Database> Consol.press) have included discussions of these longer range plans, but we shall give such discussions secondary status in this paper. We *do* wish to emphasize that we are open to suggestions concerning what the future facilities should look like; it is certain that our earliest and most energetic clients will have the greatest influence on the evolution of the system.

2. Database concepts

Before discussing the facilities to be provided in the Cedar database system, we must address the question "what properties make something a database?" We shall often contrast databases with more familiar data structures, such as Mesa record structures.

As anyone who has ever attempted to read a paper about databases knows, a large part of the database problem comes in trying to figure out what other people are saying when they talk about databases. We try to avoid jargon when possible, but nonetheless we find it useful to give a few definitions here. A *client* is a program using some facility, such as a database system. A *user*, on the other hand, is a human using a facility, such as an interactive interface to a database system. A database *query* is the operation of reading information from a database; depending on the type of interface provided, a query may be limited to reading one item or may involve thousands. An *update* is a write operation on a database.

2.1 Three basic properties of databases

Persistent

Unlike information encoded into a typical Mesa data structure, information stored in a database may last longer than one invocation of a program. This persistence of data has far-reaching consequences.

Information in a long-lived database can become more valuable than the programs that manipulate it. A database may exist long enough to be operated on by several different generations of the same application program, or to be used for applications that were not anticipated when the database was first designed. Thus the gradual evolution of the database becomes a concern analogous to the well-known problems of evolving programs.

The continued soundness of a persistent database is important. Soundness in the face of physical mishaps such as disk head crashes or tape vault fires is called *physical integrity*. A guarantee of perfect physical integrity is impossible, but a database system should reduce the probability of disaster to a level that is acceptable in light of the value of information in the database.

A much more difficult problem comes in maintaining the *logical integrity* of a database. A database is generally used to model some real-world state of affairs, such as the structure of a Mesa system, the layout of a document, or the contents of a person's appointment schedule. Logical integrity means the correct correspondence between a database and the real-world situation being modeled. A database may be damaged in a logical sense by the keying of incorrect input data, or the running of an incorrect program. These two examples should make it clear that perfect logical integrity is an ideal that cannot be guaranteed; unfortunately, the loss of confidence caused by errors in a database can be just as damaging as a catastrophic physical failure [Hoare].

One simple form of insurance for the logical integrity of a database is always provided in a database system. This is the persistent storage and enforced use of data that tells how to interpret the representation of the database. This data is somewhat analogous to the type declarations required in Mesa programs that use data structures. It should be clear that using the wrong declarations when accessing the database could lead to disaster. Some database systems go beyond this, allowing the user to state more application-specific properties that his data should satisfy. These are sometines called *integrity constraints* or *integrity assertions.*

Of course, there was long-lived information before there were any database systems: this information was stored in file systems. But file systems at best gave some assistance in preserving the physical integrity of stored data, and were no help with any of the other problems associated with persistence. Hence these problems were solved in an *ad hoc* manner by various applications.

Shared

The persistence of information in a database introduces the possibility of data sharing. This sharing is qualitatively different from the sharing that can take place among separate processes of a Mesa program.

One aspect of sharing is *consistent concurrent access* to data by programs. Here "concurrent access" does not mean that the programs' commands to read or update the database are actually executed at the same instant of time, but rather that the precise order in which these commands are executed from different programs is unpredictable.

In general, consistency of a database can only be defined with respect to an application: we say that a database state is consistent if it satisfies a certain application-specific *invariant.* For instance, an accounting system might have as an invariant the property that the total debits and credits in the system are equal. In some database systems a transaction facility is provided to help applications maintain consistency invariants. Briefly, a *transaction* is a sequence of read and write commands. The system supports the property that the entire sequence of commands executes *atomically* with respect to all other transactions, that is, the transaction executes as if no other transactions were in progress at the same time. Because there may in fact be other transactions accessing the same data at the same time, it is possible (in most systems) that two transactions may deadlock, in which case one of them must be aborted. So the price paid for concurrent access is the necessity that programs be prepared to retry aborted transactions.

An example of an application in which concurrent access capability is necessary is an interactive reservation system, say for CSL meeting rooms. By structuring the reservation system's database accesses into transactions, it should be possible to ensure that the database will remain consistent despite reservations entered concurrently from different machines on the network. The consistency invariants in this case would include having each time period for each room reserved at most once.

Another aspect of sharing is the *integration* of information into a single database. Integration means that information is kept in a consistent form in one database, even if this is not strictly necessary for the purposes of existing applications. Integration allows new problems to be solved more easily because all of the necessary information is maintained in the same database. It also encourages sharing of information at a more macroscopic level than in the reservation system example above, since it provides a set of uniform naming conventions and access methods for all users. In principle, integration makes logical integrity easier to maintain, because there is less of a tendency to duplicate data for use by different applications.

Unfortunately, sharing introduces new possibilities for logical damage to a database. For instance, one user's undebugged program can mistakenly modify the data of another user. A database's *protection system* is designed to make such damage less likely by requiring an authorization check before querying or updating the database. Some database systems provide elaborate protection facilities that give fine-grained control over access to information; this is especially necessary if sensitive information is to be shared through the database.

Large

Databases often contain large volumes of data. By one definition, a database qualifies as "large" when it becomes unacceptable to make the database inaccessible during the amount of time required to copy it.

In principle, largeness is independent of persistence. In practice, largeness generally implies persistence, but not the other way around: it is quite common to require small data objects to survive between the executions of a program. (The file Rem.cm, used by the Alto executive to store the sequence of commands remaining to be executed, is a good example).

2.2 Consequences of the properties

The discussions above were limited to immediate results of the three basic properties. We now deal with some of the more indirect consequences.

Interface independent of storage structure

Because of the shared, persistent nature of a database, its usage changes with time. To provide good performance as the access pattern shifts, it is necessary to change the database representation to keep the most common operations inexpensive. Because of the value of existing applications software, the database system should make these programs functionally immune to changes in the storage structure used in representing the database. That is, such changes should at most affect a program's performance, and not the results that it gives.

This *data independence* can be achieved by ensuring that the interface used by programmers to specify algorithms involving the database refers to a model of data, the *conceptual model,* that is a useful abstraction of stored structures (independent of their precise form). Ideally the database system would then choose a good underlying representation based on its knowldege of the user's programs, or adaptively based on execution statistics. This remains a research problem: no existing system can perform such optimizations automatically. An alternative is for the system to accept advice from the user on pragmatic matters (such as the use of internal indexes), where this advice takes the form of declarations that cannot affect the semantics of an application program.

Simple and regular structures

Another consequence of the more fundamental database properties above is that database structures should be kept simple and regular. Persistence argues for simplicity because if a structure is not simple, it is difficult for different programs to maintain it in a consistent and accurate way over time. Regular structures are easier to share for the same reason, and also because they tend to contain larger meaningful substructures, that can serve as the units for locking. With a large, irregular structure it is very likely that consecutive accesses to data will seem uncorrelated from the system's point of view, meaning that a disk access is required for each one. In more regular structures the access pattern often involves scanning data in an order that is meaningful to the system, so that caching and prefetching will increase performance.

Aggregate-level operations

By structuring a database in a very regular fashion, we make it feasible to operate on it with very concise high-level languages. This approach has been demonstrated primarily with the relational model of data [Date], which is probably the simplest model with respect to the mechanisms it provides for expressing relationships among data items. The use of such high-level languages makes it very easy to develop new applications that mainly involve queries, even when the queries use portions of the database that were developed independently. These high-level languages may also help somewhat in avoiding programming errors and maintaining the logical integrity of the database.

Multiple views

Because databases represent real-world objects and relationships, changes in the world (or in our preferred abstraction of it) may force a change in the structure of an existing database. For example, we may have to add a new field to each record of a given type, or delete an existing field. A stronger form of data independence holds when any application that isn't necessarily affected by some alteration in the conceptual model mentioned above is functionally immune to the change. (It is impossible for a nontrivial application to be immune to all possible alterations, because some of the data that it interprets may be expunged in a change of the conceptual model.) This degree of data independence requires a layer of mapping between the conceptual model and the *external model* presented to an application. Such a map is called a *view.* There may be many views that

involve the same underlying data; views also allow different external models to be used while keeping a common conceptual model. It is not generally possible to perform an update through a view unless the view is very simple.

2.3 Database versus persistent virtual memory

Is it productive to view the data structures and variables found in normal Mesa programming as database variables that don't happen to be persistent, shared, or large? To be more concrete, is the distinction between Mesa records and database objects really necessary?

To some extent this is a function of our aspirations. If we consider persistence to be the primary goal, with sharing and largeness being much less important, then we may be satisfied with a database that is essentially a long-term Mesa virtual memory. If the typing of data objects in this memory could be preserved automatically over long periods, the results would have some of the character of a database.

JaM and Smalltalk are examples of existing systems at PARC that support this kind of persistent virtual memory. Databases built in these environments can be very flexible by normal database standards. The user is able to create arbitrarily complex data structures and is able to bind procedures to individual data objects, giving a programmable view mechanism (in the sense of the section above). JaM and Smalltalk do not have a conventional database system's facilities for ensuring integrity, do not allow concurrent access (though this may change), and don't support large databases (by the standards of "BDP" applications).

2.4 Cedar database system (phase one) aspirations

Persistent

Physical integrity is aided by storing data on Juniper, and taking more frequent backups than Juniper does now (or changing Juniper to do this). Declarations live in the database, and are read using the same interface as the rest of the database. (This commonality of interface is not true for updates, for consistency reasons).

Shared

Concurrent access is provided by using page-level locking on Juniper. Currently only file-level locks are implemented. Since this does not allow the locking of logical units of data, it will perhaps not support very high degrees of concurrent access efficiently; we won't know any more about this without getting some experience. We expect the database to be used in an integrated way (although probably not at first, until confidence is built up.) The initial protection mechanism is based on file protection. This is not ideal, but the best way to improve it may be to build finer-grained protection at a level above the interface we are now developing.

Large

The database may get large in the sense that it occupies a large portion of a disk volume, but we do not expect that 24 hour real-time performance will be necessary. Thus we will be satisfied with fairly crude algorithms for responding to restructuring requests that appear to affect large volumes of data (but in principle could be handled by a more sohisticated incremental algorithm.) We do not plan to use very sophisticated algorithms to gain good performance for queries involvong huge amounts of data. We shall assume that the amount of data used to process a query will just about fit in main storage. We are primarily concerned with interactive systems rather than "DP" aplications that might involve scanning a huge database. Our performance aspirations are also limited by the bandwidth available to our file server (though we shall do local caching of file pages to reduce this traffic).

Interface independent of storage structure

This is true in our system to a large extent. The presence or absence of indexes ("inverted files") is not reflected in the statement of algorithms. Declarations are used to specify indexes, which are then maintained automatically by the system. The model does contain a form of pointer, but much of its underlying mechanism is abstracted away.

Allowed structures are simple and regular

The model is relatively simple. It contains a form of pointer, which may be tempting to abuse. One goal of this document is to explain how to represent information within this model in a well-structured way.

Aggregate-level operations

This is not part of phase one. We are implementing a relatively low-level interface, but one that seems to fit many of our applications well. We shall build aggregate operations on top of this interface.

Multiple views of data

This is not part of phase one. We plan to build views on top of this interface. The precise form of the data model that should be provided is still under review. It would be relatively easy to support the relational model, or any one of several proposed entity-based models, in an efficient way.

2.5 Cedar database applications

Several early applications of the database have been written to "shake down" the system. Some of these were one-time projects that were not intended as production systems and do not presently run with the released database system. These include a simple "report generator" written by Jim Saxe, and a database representation of components of Mesa source modules written by John Ellis. Other applications are longer-term, and are still under development at the time of this writing.

Rick Cattell is implementing a "browser" for the Cedar database system, named Palm. He has built an interim browser similar in concept to the interface described in his blue-and-white report [Cattell], except using windows and menus. The browser under design at the time of this writing is intended to run on top of Cedar user facilities, as an information management tool generalizing the functions of an executive and file system.

Both Ellis's system and the browser functionally extended the database to encode more information about the data than the current system provides. This feedback will be used in designing the phase two system's data model.

Nori Suzuki and Ray Cheng, a summer student, have designed and implemented a system to organize the CSL Notebook. This system consists of a number of programs, that process Laurel, Bravo, and Press input to the notebook and provide hardcopy listings of entries. There is not yet a program to browse or query this database on-line, although Palm will be useful for this purpose.

Eric Schmidt is building facilities for Mesa system modeling. He uses the database system to represent the relationships among the modules of a system. He has constructed some of the largest databases so far, about 20 megabytes on Juniper, providing useful feedback on the limits of Juniper and the database.

3. System design overview

Database structures and their manipulation

A *database* is a time-varying collection of data, all of which is stored as tuples. A *tuple* is an aggregate object that groups together a number of related data values. A tuple is the database analogue of a Mesa record. The following is an example of how a tuple type might be defined in Mesa, if Mesa had built-in database facilities:

```
Document: TYPE = TUPLE[title: STRING, date: LONG CARDINAL, fileName: STRING];
```

(Mesa does not have such facilities, but this Mesa-like syntax is convenient for the exposition). Here a tuple of type Document is defined to have three *attributes* (analogous to record components), with *attribute names* title, date, and fileName. Each attribute has a specific data type (STRING or LONG CARDINAL in the example). Every Document tuple has all three attributes, and different Document tuples have their own values of these attributes.

(Note that we do not allow any analogue of variant records for tuples. We also exclude the possibility of a "repeating" attribute, that is, an attribute whose value is a sequence of data values).

The database package provides operations for creating new tuples of a specified type, and for destroying existing tuples. Other operations allow the value associated with any particular attribute of a given tuple to be fetched or updated.

The *tupleset* is a data structure that is bound to the notion of tuple type. Whenever a new tuple of a particular type is created, it is entered into the tupleset associated with that type. Therefore every tuple in a database belongs to exactly one tupleset. It is sometimes useful to visualize a tupleset as a table whose rows are tuples and whose columns are attributes.

The database package supplies a way to retrieve an individual tuple or a sequence of tuples from a tupleset. The tuples to be retrieved are specified by stating allowable value ranges (or specific values) for any set of the tupleset's attributes. (The current package places some restrictions on the types of attributes allowed). If no restrictions are placed on the retrieval, all tuples in the tupleset are returned.

One further data structuring mechanism is supported in our data model. An attribute of a tuple may have type TupleRef. A value of type TupleRef is in effect a pointer to another tuple, which may be of any type. (Note that TupleRef attributes are interpreted by the database package, while other types of attributes are uninterpreted apart from the comparisons which are performed on some attributes during retrievals from a tupleset). If the tuple that is the object of a TupleRef is destroyed, that TupleRef is set to NullTuple, a distinguished value.

The package provides some specialized operations relating to TupleRef attributes. For any tuple T and any TupleRef attribute A, it is possible to retrieve the set of all tuples pointing to T via A. It is also possible to enumerate the set of all TupleRef attributes A such that one or more tuples actually point to T via A. (Note that in a good database design, the attributes that might actually contain TupleRefs to a specific type of tuple will generally be few and known in advance, but this second operation is useful for writing very general programs to manipulate databases).

TupleRef attributes are a powerful mechanism for representing relationships in a database, but are also easy to abuse. We shall present some examples of the mechanism's proper use in Section 5.

We can now contrast database tuples with Mesa records as follows. First, in the tupleset model there is a mechanism that allows all tuples of a given type to be retrieved. This means that no tuple ever becomes "garbage", since there is always an access path to it. With a Mesa record, once the last user-maintained pointer to a record disappears the record is truly inaccessible. Second, it is possible to locate all of the tuples that reference a given tuple; this is not true with pointers and Mesa records. Third, nesting of tuples is not allowed; a Mesa record may contain a component that

is also a Mesa record (not a pointer to one). Fourth, there is no analogue of variant records for tuples in the phase one implementation.

Databases and files

A database is represented as a collection of files. A file in database format is called a *segment.* A database is identified by the name of a distinguished segment, its *system segment.* A system segment contains a directory of all the other segments in the database, plus other system-related information. A new segment may be added to a database at any time by calling an interface procedure.

All of the tuples in a client-created tupleset lie in a single segment, which must be specified when the tupleset is created. This partitioning provides a crude mechanism for clustering related tuples, and may reduce the potential for conflict in concurrent accesses to a database by different applications. A default segment is provided for clients who don't care about segment structure.

The relationship between segments and concurrency is discussed further in Section 3.4; the relationship between segments and protection is covered in Section 3.5.

Data definition

The phase one system is not integrated with Mesa; the data manipulation operations sketched above are implemented as Mesa procedures in the DBTuples interface. Since the tuple is not a Mesa data type, this still leaves the problem of declarations: how do we create tuplesets, or access existing ones? The answer is to define a small number of *system tuplesets* to contain information about tuplesets and their attributes. For instance, the tupleset named TupleSetTS contains one tuple per tupleset in a database. To create a new tuple in tupleset Document, we first fetch the tuple T corresponding to tupleset Document in TupleSetTS (using the string "Document" as the retrieval key), and then pass T as a parameter to the CreateTuple procedure.

This is not the whole story, of course, since we need the representative tuple for TupleSetTS in order to retrieve the Document tuple. But there are only a small number of system tuplesets, so it is practical to provide fixed representative tuples for them in the DBTuples interface.

This technique for storing the "data dictionary" type information has the advantage that the same interface is used for data dictionary retrieval as for normal data retrieval. We do not extend the commonality of interface to updates, since the consistency of the system tuplesets with the stored data is difficult to ensure without constraints on the changes possible to the system tuplesets.

The initial version of the system is capable of maintaining indexes on various attributes of a tupleset; the information as to what indexes exist is stored in the system tuplesets. We plan to allow clustered storage of tuples related by adjacency in an index or a TupleRef attribute. We also plan to allow hints for efficient cache management.

Concurrency

The database package uses the notion of transaction (Section 2.1) to structure database accesses. Clients can ensure the logical integrity of their databases, even in the face of concurrent access, by proper grouping of queries and updates into transactions. Our notion of transaction is closely related to Juniper's. It is possible to make Juniper accesses from outside of the database package and have them included in a database transaction.

As noted above, transactions do not always succeed. Sometimes this is due to hardware or software failures, and sometimes due to conflicts in concurrent accesses. Concurrent updates to tuples in different segments almost never cause a conflict; tuplesets within the same segment are also arranged to minimize the potential for conflicts. But all clients must be prepared for the possibility of an aborted transaction. By far the simplest response to a transaction abortion is to restart the transaction from the beginning. This may be painful when the updates were performed manually; an application may wish to maintain a log for recovery of keyboard input after a transaction abort, but this must be done with care since typein may be a function of the previous data.

Protection

In the phase one system, database protection is limited to the protection provided for segments by the file system. This has its strong and weak points. On the positive side, file protection is a robust mechanism. Since it is implemented on a remote host, the malfunction of a user program cannot breach it. Users in our environment have experience with file protection and they trust it.

The drawback of file protection is its lack of flexibility in terms of the logical entities stored in a database. A TupleRef that points from my tuple to yours has to write information in the segment containing your tuple. It is not possible to create private tuples in shared tuplesets for similar reasons. These difficulties limit the integration that can be achieved among users who don't share write access to database files.

Utilities

Database dump / reload. Dumping a database in an Ascii form and reloading it are functions that must currently be provided by each application, although these facilities do exist for databases in the form assumed by the browser.

Database creation -- individual databases. We support private databases. We shall promote formation of integrated CSL databases where this seems appropriate.

Database scavenger. We intend to provide a tool for diagnosing and repairing broken databases. Databases may break because of bugs in the system, or due to the use of unsafe language features. We haven't yet, largely because the information involved spans all levels of the database.

Environmental requirements

The system uses Juniper to store the database. Other bases for storage are under experimental development; IFS/Leaf access is tentatively available. All of the database software runs on the client machine; thus the system's speed may depend a great deal upon the amount of main memory available on the client machine for caching database pages, as well as on the raw speed of the client machine.

4. Client interface

This section documents the interface provided by DBTuples.mesa (stored on [Ivy]<CedarDB>System>).

The reader should keep in mind that as Cedar language facilities become available, the types in this interface will change. Comments relating to Cedar appear below in fine print. The final interface will be "safe" in the Cedar sense; in particular, all POINTERS in the current interface will become REFS.

4.1 Types

In this subsection we shall describe only the most important types defined in the interface. Less pervasive types are treated below at the point where they are first used.

```
Tuple: TYPE = POINTER TO OpaqueTuple;
OpaqueTuple: TYPE = PRIVATE RECORD[ ... ];
```

A Tuple is not a database tuple; it is a *handle* for a database tuple. All accesses to database tuples are performed by calling interface procedures with Tuple parameters. (In Cedar, Tuple will be an opaque type).

It is pointless to save a Tuple in the database, since a Tuple is only valid during a single invocation of the system (more precisely, between calls to OpenDatabase and CloseDatabase below). Comparing two Tuples for equality of their underlying database tuples requires a call to an interface procedure. The fact that a Tuple has a main-storage data structure associated with it means that clients are discouraged from building up large data structures with Tuple components; large structures should be represented in the database itself. (Until the Cedar garbage collector is used with the database system, Tuples will be allocated from heap storage and the programmer will have the responsibility of explicitly freeing Tuples that are no longer needed. For this reason the system always constructs a new OpaqueTuple when the user asks for a Tuple, even if a Tuple already exists for that database tuple.)

```
Value: TYPE = RECORD[
 SELECT type: ValueType FROM
  tupleRef => [value: Tuple],
  boolean => [value: BOOLEAN],
  enumerated => [value: UNSPECIFIED],
  cardinal => [value: CARDINAL],
  integer => [value: INTEGER],
  longCardinal => [value: LONG CARDINAL],
  longInteger => [value: LONG INTEGER],
  string => [value: STRING],
  record => [value: POINTER TO UNSPECIFIED],
  arrayDescriptor => [value: DESCRIPTOR FOR ARRAY OF UNSPECIFIED]
  ENDCASE];

ValueType: TYPE = {tupleRef, boolean, enumerated, cardinal, integer, longCardinal,
longInteger, string, record, arrayDescriptor};
```

Storing Mesa data values in tuples presents several problems. First, since we would like to define a single operation to store a new value into a specified attribute of a tuple (for instance), there must be a single type for all values that pass through this "store-value" procedure. This is the type Value above. Translation to and from this variant record representation is clumsy in some cases, but it seems preferable to the alternatives. This problem will last for as long as the database package is just a package, and not integrated with Mesa. (The REF ANY type in Cedar Mesa should help). In the meantime, we provide a set of tersely named type conversion routines described in Section 4.3.

The second problem is that in the runtime environment of a Mesa program there is no information about types, particularly user-defined types (this will change with Cedar). For this reason, the phase one database package is limited to storing only a few common Mesa datatypes with type safety. These types are enumerated in ValueType above. The alternatives boolean, cardinal, integer, longCardinal, longInteger, and string correspond to the Mesa types with these names, while arrayDescriptor corresponds to DESCRIPTOR FOR ARRAY OF UNSPECIFIED. The enumerated and record alternatives have no type-safe interpretation; they correspond, respectively, to an UNSPECIFIED and a POINTER TO UNSPECIFIED that really points to some fixed-length structure. The type corresponding to tupleRef is simply Tuple.

A final problem with data values is that there is no standard format for representing Mesa values on external media (this is a recognized problem for Cedar, but nobody is working on it now). We just store the bits as we find them. A client who stores a Mesa record in a database is well-advised to make it MACHINE DEPENDENT.

4.2 System tuplesets

```
TupleSet: TYPE = Tuple; -- from TupleSetTS
TupleSetTS: READONLY TupleSet;
```

The *tupleset tupleset* is a system-defined tupleset containing one tuple per user-defined tupleset. This tuple acts as a representative for the tupleset in various interface procedures defined below, such as the procedure to create a new tuple in a specified tupleset, and therefore a Tuple for such a tuple is called a TupleSet. The representative for the tupleset tupleset itself is TupleSetTS, and appears as an interface variable in DBTuples. (TupleSetTS is a system tuple, not a dictionary tuple; it isn't actually stored in any database).

```
Attribute: TYPE = Tuple; -- from AttributeTS
AttributeTS: READONLY TupleSet;
```

The *attribute tupleset* is a system-defined tupleset containing one tuple per attribute of user-defined tuplesets. This tuple acts as a representative for the attribute in various interface procedures defined below, such as the procedure to fetch the value associated with a specific attribute of a given tuple, and therefore a Tuple for such a tuple is called an Attribute. The representative for the attribute tupleset itself is AttributeTS, and appears as an interface variable in DBTuples.

```
tsOwner: READONLY Attribute;
 -- string: login name of owner (empty string means system)
tsName: READONLY Attribute;
 -- string: name of tupleset
tsSegment: READONLY Attribute;
 -- string: name of segment containing tupleset
tsIsDomain: READONLY Attribute;
 -- boolean: [true, false] => permit TupleRefs [to, from] the tupleset's tuples
```

TupleSetTS has four attributes, shown above. The complete name of a tupleset is an ordered pair of strings, the first being the login name of the "owner" of the tupleset, and the second being an arbitrary string describing the tupleset. These names are unique within a single database: the intent of this mechanism is to reduce the possibility of inadvertent naming conflicts among users. All of the tupleset's tuples reside in the same segment, whose name is given by tsSegment. The tsIsDomain attribute indicates whether the tupleset's tuples are *entities*, which may be referenced by but may not reference other tuples, or *relationships*, which may reference but may not be referenced by other tuples. This attribute is not enforced at present, and is only used by a few programs; it is further described in Section 5.

```
aName: READONLY Attribute;
 -- string: name of attribute
aTupleSet: READONLY Attribute;
 -- tupleRef (to a TupleSetTS tuple): tupleset to which attribute belongs
aType: READONLY Attribute;
 -- cardinal (from ValueType): type of attribute
aLength: READONLY Attribute;
 -- cardinal: length of record or length hint of string/arrayDescriptor
aTSHint: READONLY Attribute;
 -- tupleRef: for tupleRef-valued attributes only, a hint at the tupleset referenced
```

AttributeTS has four attributes, shown above. The name of an attribute is local to the attribute's tupleset; that is, aName above must be distinct for all of the attributes of a tupleset, but not necessarily for all of the attributes in a database. The attribute aTupleSet identifies the attribute's tupleset, i.e. its tuple in TupleSetTS, while aType gives the attribute's type (as described in the previous section). Finally, aLength describes attributes of type record, string, and arrayDescriptor more fully. When aType = record, aLength gives the number of words in the record. When aType = string, aLength gives the "length hint", which is the number of bytes that can be stored directly in the body of a database tuple. When a string exceeds this length a linked representation is used for the string's tail, at a cost of several extra words. So when a good estimate of the length of a string attribute can be made, it may pay to specify it. Note that even when aLength = 0, strings of any length may be stored. When aType = arrayDescriptor, aLength gives the "length hint" in array elements (words). The aTSHint attribute is used iff aType = tupleRef; the client may use it to indicate which tupleset's tuples the attribute references. This is not enforced, and aTSHint currently defaults to NIL (no hint).

The system-defined tuplesets that describe indexes are generally of little interest to clients, since the semantics of database operations are independent of the existence of indexes. We describe these two tuplesets now primarily as a means of delineating the kinds of indexes that the system supports.

Indexes are maintained for tuplesets; that is, an index contains entries for all of the tuples in a specified tupleset. A tuple is indexed according to a key that is a sequence of values of certain of the tuple's attributes. This set of attributes and their ordering is part of the specification of an index. Key comparisons are performed in the usual lexicographic fashion: If the first components of two value sequences are unequal then the sequence with the larger first component is larger, otherwise the tails of the two sequences are compared using the same algorithm. Individual components are compared using built-in comparison functions for each value type. (At present the system supports indexes only on values of type string, longCardinal, longInteger, cardinal, integer, and tupleRef). This definition of comparison is of interest to clients not because of indexes *per se*, but because one of the tuple retrieval operations to be described in a later section uses the same definition. (Indexes on tupleRef-valued fields are not useful for their ordering, but for improving performance for equality comparisons on multiple tupleRef-valued fields.)

```
IndexTS: READONLY TupleSet;
iTupleSet: READONLY Attribute;
 -- tupleRef (to a TupleSetTS tuple): tupleset to which index belongs

IndexFactorTS: READONLY TupleSet;
ifIndex: READONLY Attribute;
 -- tupleRef (to a IndexTS tuple): index to which factor belongs
ifOrdinalPosition: READONLY Attribute;
 -- cardinal: significance of factor in determining key ordering
ifAttribute: READONLY Attribute;
 -- tupleRef (to an AttributeTS tuple): attribute that factor includes in key for index
```

A tuple in IndexTS represents an index, and the only attribute of this tuple is the tupleset that the index indexes (iTupleSet). An index's notion of key is defined by a collection of tuples in IndexFactorTS. An *index factor* specifies the way in which one attribute participates in defining the

index's key. The factor's index is given by ifIndex, and its attribute by ifAttribute. The ordering among these attributes is specified by the ifOrdinalPosition, with 1 being the most significant, 2 the next, and so on.

That completes our description of the four system-defined tuplesets. These tuplesets contain all of the accessible data about the user-defined data in a database. A client program can read these tuplesets using the same interface that it uses for all other data. Attempts to perform updates through this interface results in an ERROR; all updates are performed implicitly by CreateTupleSet, CreateAttribute, etc. below in order to ensure the consistency of this data. (We acknowledge the fact that as the system evolves these procedures will have to grow more elaborate).

4.3 Procedures

Manipulating individual tuples

```
CreateTuple: PROCEDURE[ts: TupleSet, avl: AttributeValueList← EmptyList] RETURNS[t: Tuple];
DestroyTuple: PROCEDURE[t: Tuple];
AttributeValueList: TYPE = DESCRIPTOR FOR ARRAY OF RECORD[attribute: Attribute, value: Value];
EmptyList: DESCRIPTOR FOR ARRAY OF UNSPECIFIED = DESCRIPTOR[NIL, 0];
```

CreateTuple creates a new tuple t in tupleset ts, with fields initialized as specified by avl. Attributes of ts that are omitted from avl are given null values (0 for numbers, "" for strings, NullTuple for tupleRefs). Such a multiple assignment can go faster than a sequence of normal assignments, since any indexes on the tupleset are updated only at the end. DestroyTuple destroys tuple t; any tupleRefs that point to t are changed to NullTuple.

```
SetF: PROCEDURE[t: Tuple, a: Attribute, v: Value];
SetMultipleF: PROCEDURE[t: Tuple, avl: AttributeValueList];
GetF: PROCEDURE[t: Tuple, a: Attribute] RETURNS[v: Value];
TS: READONLY Attribute;
```

SetF sets the value of t's attribute a to v, and GetF retrieves the previously-set value. SetMultipleF has the effect of a sequence of SetF calls on t, with attributes and values specified in avl. (The "F" in these names is an anachronism, which stood for "field"). TS is a special attribute such that for any Tuple t, GetF[t, TS] returns t's TupleSet.

```
Eq: PROCEDURE[t1, t2: Tuple] RETURNS[BOOLEAN];
Null: PROCEDURE[t: Tuple] RETURNS[BOOLEAN];
```

Eq returns TRUE iff the same database tuple is referenced by t1 and t2. This is *not* equivalent to the Mesa expression "t1 = t2". Null returns TRUE iff its argument is equal to the distinguished NullTuple.

```
ReleaseTuple: PROCEDURE[t: Tuple];
ReleaseValue: PROCEDURE[v: Value];
```

The above routines are required until the garbage collector is used. All string, tuple, record, and arrayDescriptor values obtained from a GetF or equivalent myst be released. ReleaseTuple tells the database system that t is the last client reference to its TupleObject, so the TupleObject can be reused. ReleaseValue works on tupleRef values as well as strings and arrayDescriptors.

Type conversion routines

```
T2V: PROCEDURE[t: Tuple] RETURNS[v: Value];
B2V: PROCEDURE[b: BOOLEAN] RETURNS[v: Value];
E2V: PROCEDURE[e: UNSPECIFIED] RETURNS[v: Value];
C2V: PROCEDURE[c: CARDINAL] RETURNS[v: Value];
I2V: PROCEDURE[i: INTEGER] RETURNS[v: Value];
LC2V: PROCEDURE[lc: LONG CARDINAL] RETURNS[v: Value];
LI2V: PROCEDURE[li: LONG INTEGER] RETURNS[v: Value];
S2V: PROCEDURE[s: STRING] RETURNS[v: Value];
R2V: PROCEDURE[r: POINTER TO UNSPECIFIED] RETURNS[v: Value];
A2V: PROCEDURE[a: DESCRIPTOR FOR ARRAY OF UNSPECIFIED] RETURNS[v: Value];
```

The x2V routines convert the various Mesa types to Values. It is recommended that these routines be used rather than explicit variant record constructors as the representation of Values may change.

```
V2T: PROCEDURE[v: Value] RETURNS[Tuple];
V2B: PROCEDURE[v: Value] RETURNS[BOOLEAN];
V2C: PROCEDURE[v: Value] RETURNS[CARDINAL];
V2I: PROCEDURE [v: Value] RETURNS[INTEGER];
V2LC: PROCEDURE [v: Value] RETURNS[LONG CARDINAL];
V2LI: PROCEDURE [v: Value] RETURNS[LONG INTEGER];
V2S: PROCEDURE [v: Value] RETURNS[STRING];
V2R: PROCEDURE [v: Value] RETURNS[POINTER TO UNSPECIFIED];
V2A: PROCEDURE [v: Value] RETURNS[DESCRIPTOR FOR ARRAY OF UNSPECIFIED];
V2E: PUBLIC PROCEDURE [v: Value] RETURNS[UNSPECIFIED];
```

The V2x routines convert Values to the various Mesa types. The IncompatibleValueType ERROR is raised if the value is of the wrong type. It is recommended that these routines be used rather than explicit field extraction as the representation of Values may change.

Special convenience procedures such as those below may be added to the interface as they seem appropriate, but these are experimental styles of database access and are likely to change.

```
GetFS: PROCEDURE[t: Tuple, a: Attribute] RETURNS[v: STRING];
SetFS: PROCEDURE[t: Tuple, a: Attribute, v: STRING];
```

GetFS ("GetF-string") is legal for any attribute a of a tuple t. It converts boolean, cardinal, integer, longCardinal, and longInteger valued attributes to strings before returning them; it raises the IncompatibleAttributeType ERROR for record and arrayDescriptor attributes. When a is a tupleRef attribute, GetFS returns the string value of t's first-defined attribute. (This is consistent with a convention that the first attribute of each tuple is its "name"). SetFS is the inverse of GetFS, but is not defined for tupleRef attributes.

Searching for tuples

The interface provides two forms of retrieval from a tupleset that yield a collection of tuples. A result of the first form, called a MatchSet, contains all tuples whose attribute values fall in specified ranges. A result of the second form, called a RefSet, contains all tuples with a specific tupleRef attribute pointing to a particular tuple. (We may unify these two kinds of sets. The thing we will not do at this interface is provide additional operations on such sets, since this quickly blossoms into a general query language).

MatchSets and RefSets are implemented as generators. An initial call is used to define the query, and this returns the MatchSet or RefSet. Then succeeding calls to the "next" operator are used to scan through the elements of the set. When the set is empty, NullTuple is returned.

Because the sets are not materialized at the time of the initial call, it is possible for the sets to gain or lose members during a scan. For instance, a SetF to a tupleRef attribute could remove a tuple from an existing RefSet. When a tuple is added to a set while a scan is in progress, it is

unpredictable whether or not it will be seen by the scan. (At present, if an index exists on the attributes specified for a MatchSet then the newly added tuples appear iff its key is greater or equal to the key of the tuple most recently returned; we may implement this rule more uniformly later).

At the end of a scan, the MatchSet or RefSet should be explicitly released, to allow the system to clean up internal data structures associated with the scan. Any use of a released MatchSet or RefSet is an error. (The release routines will probably still be available in Cedar for performance reasons, but their use will be optional since the garbage collector will eventually reclaim any unused scans).

```
ForMatchSet: PROCEDURE[ts: TupleSet, tm: TupleMatch← NIL] RETURNS[ms: MatchSet];
NextMatch: PROCEDURE[ms: MatchSet] RETURNS[t: Tuple];
ReleaseMatchSet: PROCEDURE[ms: MatchSet];
TupleMatch: TYPE = DESCRIPTOR FOR ARRAY OF AttributeMatch;
AttributeMatch: TYPE = RECORD[
 attribute: Attribute,
 lowValue: Value,
 highValue: Value← NIL];
MatchSet: TYPE = POINTER TO OpaqueMatchSetObject;
```

A tuple in ts belongs in the result of ForMatchSet if the values of its attributes listed in tm, when considered collectively as a "key" in the sense of Section 4.2, lie within the specified inclusive range. If omitted, highValue is assumed to be the same as lowValue. The attributes referenced by the MatchSet may not be of type record or arrayDescriptor. In the current system, the tuples are enumerated in order iff there exists an index on a subset or superset of the attributes referenced in the MatchSet. If there is an index on a subset of the attributes, the ordering will be only on the subset. If there is more than one index including the attributes, it is undefined which will be used. Other additions are anticipated, e.g. half-open intervals on the lowValue or highValue side. Warning: tm is currently not copied, and thus must not be deallocated by the user until done with ms).

```
ForRefSet: PROCEDURE[t: Tuple, a: Attribute] RETURNS[rs: RefSet];
NextRef: PROCEDURE[x: RefSet] RETURNS[y: Tuple];
ReleaseRefSet: PROCEDURE[rs: RefSet];
RefSet: TYPE = POINTER TO OpaqueRefSetObject;
```

A tuple belongs in the result of ForRefSet if the value of its tupleRef attribute a is t. (In the current system, tuples are materialized in order of the time when attribute a was set to t; later we shall add an ordering spec to this procedure).

```
GetAllRefAttributes: PROCEDURE[t: Tuple] RETURNS[al: AttributeList];
AttributeList: TYPE = DESCRIPTOR FOR ARRAY OF Attribute;
```

Returns a list of all tupleRef attributes a such that some tuple points to t with a. That is, GetAllRefAttributes returns the attributes a such that ForRefSet[t, a] returns a nonempty set.

For convenience, the interface provides a procedure for retrieving a tuple that can be uniquely identified by some of its attribute values.

```
FetchTuple: PROCEDURE[ts: TupleSet, tm: TupleMatch, version: Version← OldOnly] RETURNS[t:
Tuple];
Version: TYPE = {OldOnly, NewOnly, NewOrOld};
```

Returns the unique tuple in ts that matches tm, i.e. has the indicated values of the given attributes. If version = OldOnly, raises (ERROR) MultipleMatches if more than one tuple in ts matches tm; otherwise, it *creates* a new tuple in ts with attributes values given by [the lowValues in] tm. Raises (ERROR) NoMatches if no tuple in ts matches tm.

The following program fragment is an example of FetchTuple in use:

```
owner: STRING ← "CSL";
name: STRING ← "Document";
tm: ARRAY [0..2) OF AttributeMatch ← [[tsOwner,S2V[owner]],[tsName,S2V[name]];
```

```
docTS: TupleSet ← FetchTuple[TupleSetTS, @tm];
```

The last statement fetches the TupleSet CSL.Document from the tupleset tupleset, using the fixed system tuples TupleSetTS, tsOwner, and tsName.

Defining tuplesets

Every time that a client prepares to access a database, it must either call the system to create the tuplesets and attributes it will use, or call the system to fetch already existing tuplesets and attributes. The logic involved in the two operations is often quite similar (the main difference being that in the first case the client may wish to create indexes, while in the latter situation indexes are probably irrelevant). With this in mind, we allow a version parameter to CreateTupleSet and CreateAttribute below. By using version = NewOnly when creating, and version = OldOnly when not creating, the same code can be used to initialize the client in either case. (Note that version defaults to NewOnly, justifying the names of these procedures).

```
CreateTupleSet: PROCEDURE[
 name: STRING, owner, segment: STRING← NIL,
 isDomain: BOOLEAN← TRUE,
 expectedNTupleRefs: CARDINAL← 0,
 version: Version← NewOnly] RETURNS[TupleSet];
```

CreateTupleSet either defines a new tupleset or retrieves an existing tupleset. In either case the name, owner, segment, and isDomain parameters correspond to the tsName, tsOwner, tsSegment, and tsIsDomain attributes of TupleSetTS (described in Section 4.2). ExpectedNTupleRefs is a hint at the maximum number of different tupleRef attributes that will reference tuples in the tupleset (used to allocate space for pointers). If version = NewOnly or NewOrOld, CreateTupleSet creates a new tupleset with the given owner and name in the given segment; the owner defaults to "" (meaning public), and the segment defaults to a common public segment. If version = OldOnly or NewOrOld, CreateTupleSet searches for the existing tupleset with the given owner and name (segment is ignored), and returns it. CreateTupleSet raises (ERROR) DictionaryTupleExists if a tupleset with this owner and name already exists and version = NewOnly, and raises (ERROR) NoMatch if a tupleset with this owner and name does not exist and version = OldOnly. The user may create unnamed tuplesets by giving the empty string as a name; this is the only case where the name need not be distinct from all other tuplesets with the same owner. Since it is not possible to distinguish unnamed tuplesets by name, the client normally saves the tupleset tuple in a global variable and destroys the tupleset after program execution.

```
CreateAttribute: PROCEDURE[ts: TupleSet, name: STRING, type: ValueType,
 length: CARDINAL← 0, tsHint: TupleSet, version: Version← NewOnly] RETURNS[Attribute];
```

CreateAttribute either defines a new attribute or retrieves an existing attribute of the tupleset ts. The version parameter works as in CreateTupleSet, and the same ERRORs are raised. The name, type, tsHint, and length parameters correspond to the aName, aType, aTSHint, and aLength attributes of AttributeTS (described in Section 4.2). At present, CreateAttribute may only be called if ts has no tuples in it.

```
CreateIndex: PROCEDURE[ts: TupleSet, order: TupleOrder, version: Version← NewOnly]
RETURNS[Index];
TupleOrder: TYPE = DESCRIPTOR FOR ARRAY OF Attribute;
```

Creates an index on tupleset ts, using the sequence of attributes in order to define the key (as described in Section 4.2). The version parameter works as in CreateTupleSet, and the same ERRORs are raised. At present, CreateIndex should only be called if ts has no tuples in it.

```
DestroyTupleSet: PROCEDURE[ts: TupleSet];
```

```
DestroyAttribute: PROCEDURE[a: Attribute];
DestroyIndex: PROCEDURE[i: Index];
```

DestroyTupleSet destroys the tupleset ts, its attributes and indexes, and all of its tuples. DestroyIndex just destroys the index i (which must have been obtained either from CreateIndex (unlikely) or by retrieval from IndexTS. DestroyAttribute is not implemented, and is likely to remain so for some time; it destroys the attribute a, removing it from all tuples in a's tupleset.

Starting up; transactions

The role of a transaction mechanism was described in the introduction. One requirement is that other packages used in conjunction with the database can group their operations into the same transaction when this is appropriate. Of course, when this is done the interactions among packages must be understood, since one package might commit the transaction while the other had some persistent data structure in an inconsistent state.

This is how the database package uses Juniper transactions. When a OpenDatabase starts the database session, a transaction is either passed in from the outside world, or created by the database package (the latter is a convenience feature). The operation MarkTransaction below forms the end of a database transaction and the start of a new one; this is implemented by executing a "checkpoint" (commit and hold read locks) on Juniper [DFS]. The operation CloseDatabase first does MarkTransaction, then kills the Juniper transaction if it was created by the package.

Clients must decide when to tell the system that a transaction is complete, and must be prepared to deal with unsolicited notification that the current transaction has been aborted because of system failure or lock conflict. Note that when the database package cooperates with other packages, a database transaction may be forced to abort because of the failure of some activity in the other package that is part of the same transaction.

There is only one transaction per instance of the database software. This is a major simplification of the package itself, and incidentally removes the need to pass a TransactionID on each call to most procedures in the DBTuples interface.

```
CreateDatabase: PROCEDURE[
 useTransID: Transaction← NIL,
 userName, password, databaseName, server: STRING← NIL,
 nCachePages: CARDINAL ← DefaultNCachePages,
 nBytesInitial: LONG CARDINAL← DefaultNBytesInitial,
 nBytesPerExtent: LONG CARDINAL← DefaultNBytesPerExtent]
 RETURNS[transID: Transaction, welcomeMsg: STRING];
OpenDatabase: ... ;
CloseDatabase: PROCEDURE;
Transaction: TYPE = UserPineDefs.Transaction;
```

If transID is NIL then CreateDatabase establishes a new connection with Juniper (Juniper login), creates a new empty database, and returns the resulting Transaction. If transID is non-NIL it stores transID internally instead of creating a new transaction. The userName, password, and server are used for logging in; the databaseName is used as described below. nCachePages, nBytesInitial, and nBytesPerExtent are used as hints to improve performance: they give the number of pages to use for the local cache, the initial length in bytes of database segments, and the incremental size for lengthening database segments, respectively. OpenDatabase is identical to CreateDatabase, except it starts with a previously existing database and does not use the nBytesInitial and nBytesPerExtent arguments. CloseDatabase ends the final database transaction. If the enclosing Juniper transaction was created by OpenDatabase it then does a Juniper logout. So every client program has the form: OpenDatabase[...]; -- database operations, including zero or more MarkTransaction calls -- CloseDatabase[];

The names of the files containing the database have the form databaseName$segmentName.SEGMENT, where segmentName is the string passed to CreateTupleSet and databaseName is the string passed to OpenDatabase. The database name should contain the directory on Juniper where the database is to reside; if databaseName is defaulted, it is given the value "<CedarDB>Database". The system segment (where database system information is kept) and the common segment (where the user data and tupleset information are kept by default) are named SYSTEM and COMMON, respectively; thus for the default common database these Juniper files are <CedarDB>Database$SYSTEM.SEGMENT and <CedarDB>Database$COMMON.SEGMENT. *It is a very bad idea to play with database files other than through the database interface.*

CreateSegment: PROCEDURE[segmentName: STRING, version: Version← NewOnly];

CreateSegment either adds a new segment to the currently open database or verifies the existence of an existing segment. CreateSegment raises (ERROR) SegmentExists if a segment with this name already exists and version = NewOnly, and raises (ERROR) NoSegmentExists if no segment with this name is present and version = OldOnly.

AbortTransaction: PROCEDURE;

Revokes all SetFs, CreateTuples since startup or last MarkTransaction. The effect on the database is as if the transaction had never been started. OpenDatabase may be called following AbortTransaction.

TransactionHasAborted: PROC [] RETURNS [BOOLEAN];

Returns TRUE if transaction has spontaneously aborted. The client is responsible for calling AbortTransaction after finding TransactionIsAborted = TRUE. Note: polling for aborted transactions is only required if no calls on DBTuples generate Juniper accesses. If such a call is made after a spontaneous abort, the signal DBException.Fatal[TransactionReset] will normally result.

MarkTransaction: PROCEDURE;

This commits the current database transaction, and immediately starts a new one.

5. Database design

This section is a discussion of database design: the process of representing abstractions of real-world information structures in a database. This is an important, poorly understod topic, just as the design of programs is; don't expect any startling revelations in these pages. To some extent the discussion here is specialized to the data structures available in the Cedar database package. In particular, we shall take every opportunity to show the weaknesses of the current design, and will give some indication of how things may change to correct the observed faults.

What are the properties of well-designed database? To a large extent these properties follow from the general properties of databases, as discussed in Section 2. For instance, we would like our databases to extend gracefully as new types of information are added, since the existing data and programs are likely to be quite valuable.

It may be useful to consider the following point. The distinguishing aspect of information stored in a database system is that at least *some* of it is stored in a form that can be interpreted by the system itself, rather than only by some application-specific program. Hence one important dimension of variation among different database designs is in the amount of the database that is system-interpretable, i.e. the kinds of queries that can be answered by the system.

As an example of variation in this dimension, consider the problem of designing a database for organizing a collection of Mesa modules. In the present Mesa environment, this database would need to include at least the names of all the definitions modules, program modules, configuration descriptions, and current .Bcd files. A database containing only this information is little more than a file directory, and therefore the system's power to answer queries about information in this database is very limited. A somewhat richer database might represent the DIRECTORY and IMPORTS sections of each module as collections of tuples, so that queries such as "does module X import interface Y?" can be answered by the system. This might be elaborated further to deal with the use of individual types and procedures from a definitions module, and so on. There may be a limit beyond which it is useless to represent smaller objects in the database; if we aren't interested in answering queries like "what procedures in this module contain IF statements?", it may be attractive to represent the body of a procedure (or some smaller naming scope) as a Mesa data structure that is not interpretable by the database system, even though it is stored in a database.

We shall illustrate a few more database design ideas with the following problem: design a database of information about a collection of documents. Our current facilities, which again are simply file directories, leave much to be desired. The title of a document on the printed page does not tell the reader where the document is stored or how to print a copy. Relationships between different versions of the same basic document are not explicit. Retrievals by content are impossible. Our goal here is not to solve all of these problems, but to start a design that has the potential of dealing with some of them.

Each document has a title and a set of authors. Hence we might represent a collection of documents as a tupleset:

```
Document: TYPE = TUPLE[title: STRING, authors: STRING, fileName: STRING];
```

Here the authors' names are concatenated into a single string, using some punctuation scheme to allow the string to be decoded into the list of authors. This is a very poor database design because it does not allow the system to respond easily to queries involving authors; the system cannot parse the encoded author list. Since the number of authors is variable, a much better design involves two tuplesets:

```
Document: TYPE = TUPLE[title: STRING, fileName: STRING];
DocAuthor: TYPE = TUPLE[document: TUPLEREF--to Document--, author: STRING];
```

Here we have one Document tuple per document, plus, for each document, one DocAuthor tuple per author of that document. Each DocAuthor tuple points to its Document tuple via the TUPLEREF attribute document. Now a search of the author attribute of the DocAuthor tupleset can be used to locate all documents authored by a given person.

Documents have other interesting attributes. One is the "class" of document, e.g. internal memo, blue-and-white report, or journal article. Another is the date on which the document was produced. Since these are single-valued attributes that depend functionally on the document, they can be included in the Document tupleset in an obvious way:

```
Document: TYPE = TUPLE[title: STRING, date: LONG CARDINAL, class: CARDINAL, fileName: STRING];
DocAuthor: TYPE = TUPLE[document: TUPLEREF--to Document--, author: STRING];
```

A few tuples from a database of documents, defined according to these two tuplesets, are shown graphically in Figure 1.

Already some weaknesses in the database system's present type structure are evident. It should be possible to declare that document points to a Document tuple; right now we can only say this in a comment. It should be possible to use enumerated types in the database, instead of encoding them as CARDINAL (see class above). Part of the problem here is that current Mesa redefines such types whenever the definitions module containing them is recompiled, and certainly whenever new elements are added to the enumeration; this must be changed before such types can be useful in the database. Finally, it should be possible to store other user-defined types such as TimeDefs.PackedTime in the database, and have them displayed to the user as properly-formatted strings when the user looks at them with a browser. This requires the attachment of user-written Mesa procedures to database attributes, which in turn requires the ability to fetch a Mesa procedure from a database and execute it.

The above database design has one property that may be viewed as an advantage or a drawback: authors are strings, so anything is acceptable as an author. This weak typing gives a lot of flexibility: authors may be people or institutions, and the database will never complain that it doesn't know the author you just attached to a certain document. The corresponding drawback, as you might expect, is that the system is not helpful in catching errors when a new document is added to the database. If "Cattell" is mistakenly spelled "Catell", then one of Rick's papers will not be properly retrieved by a later search.

A step in the direction of stronger type checking is to provide a separate tupleset for authors, and represent the author attribute of DocAuthor as a TUPLEREF to an Author tuple. In database jargon, the Author tuple "Cattell" is an *entity*. Document tuples are also entities, and DocAuthor tuples represent a *relationship* between documents and authors.

```
Document: TYPE = TUPLE[title: STRING, date: LONG CARDINAL, class: CARDINAL, fileName: STRING];
DocAuthor: TYPE = TUPLE[document: TUPLEREF--to Document--, author: TUPLEREF--to Author--];
Author: TYPE = TUPLE[name: STRING];
```

Figure 2 shows the database of Figure 1, cast in to this "entitized" form.

What are the problems with entities? One is in coping with orthogonal attributes of entities. It is certainly inelegant to have one "entity" represent Cattell as an author, and another represent him as a member of CSL. So we need a type system that is flexible enough to allow some authors to be from CSL, and some not (otherwise we must stop co-authoring papers with non-CSLers!) Similar comments apply to having organizations as authors. Both PIE and Flavors address themselves to the orthogonal attributes problem, in different ways; our system will have to find a solution that is appropriate to its needs.

Another limitation of our current system is that entities must be referenced with TUPLEREF attributes. We might like to be able to declare that a certain tupleset is an "entity tupleset", and declare a certain attribute to be the "name" of an entity. The system could then (1) guarantee that the names of entities within a single tupleset remain unique, and (2) support a new type, say "ENTITY REF", similar to TUPLEREF but with alternative implementations, e.g. storing the entity name.

To make certain operations more convenient and allow for future compatibility, some provision has been made in the current system for representing entity tuplesets. On creation of a tupleset, the user *may* specify [through the tslsDomain argument] whether the tupleset represents entities or relationships. This information is *not* currently used or enforced by the database system itself, but

by certain utilities such as the browser and Ascii dump/load programs, which interpret them as follows:

Entity tuplesets represent types of objects. The first attribute of every entity tupleset (the one for which CreateAttribute is first called) must be of type string, and is treated as a name which uniquely distinguishes every tuple in the tupleset. Entity tuples may have other attributes, too, at present, but these attributes should not be of type TupleRef. An example of an entity tupleset is the Document tupleset.

Relationship tuplesets represent relations between entities. They have no constraints on the types or interpretation of their attributes, except that TupleRef attributes may only reference entity tuples, not relationship tuples. An example of a relationship tupleset is the DocAuthor tupleset.

Upon creation of a TupleRef attribute, the client may give a tsHint argument used by some utility programs but not currently enforced by the system. The tsHint specifies the TupleSet whose tuples the attribute may reference. If tsHint is defaulted, the attribute may reference any [entity] tuple.

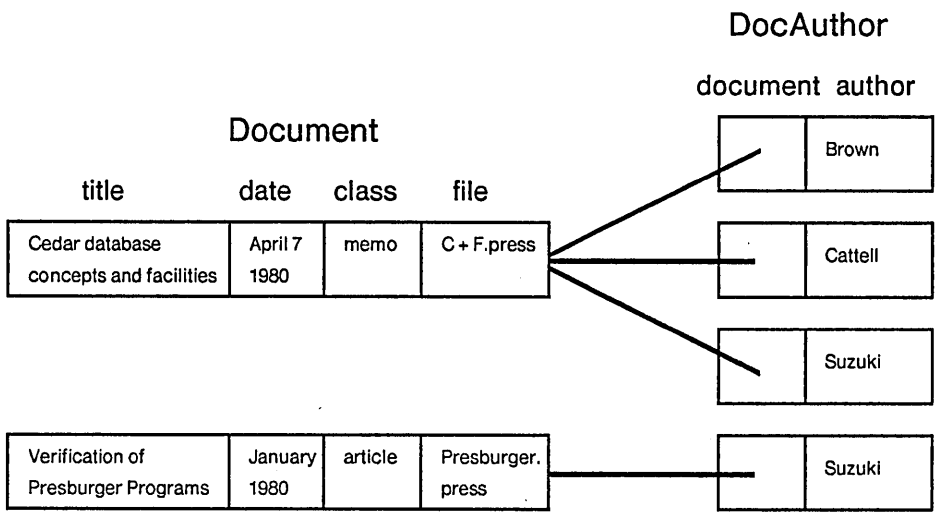


Figure 1. A fragment of a simple database for documents

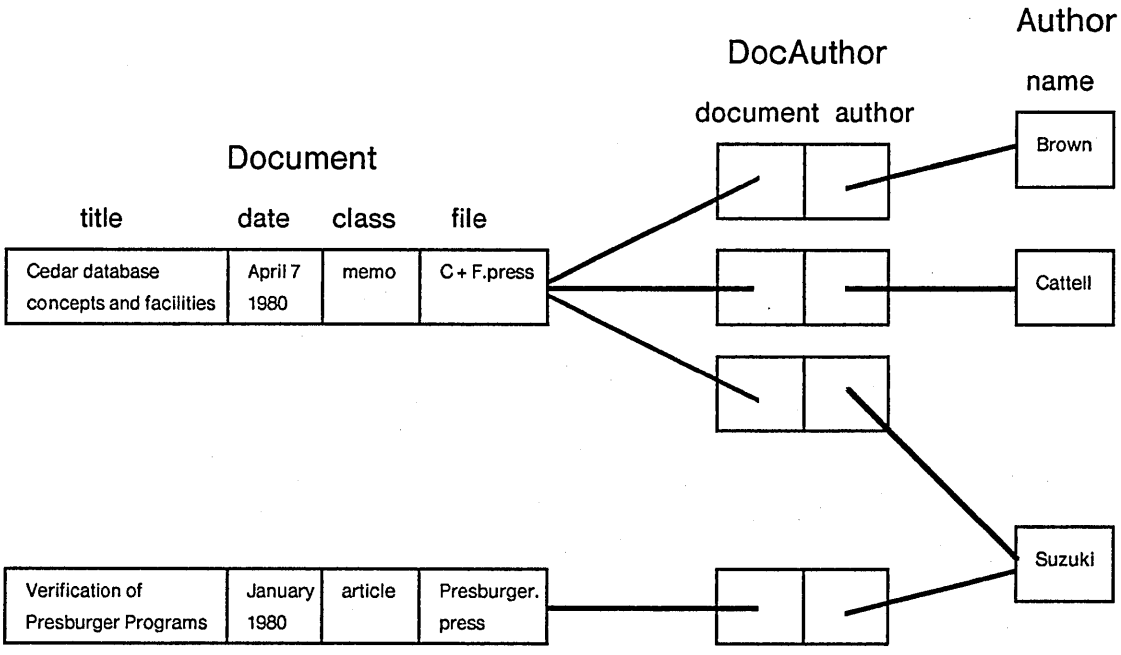


Figure 2. The same database with authors as entities

What other information should be present in our document database? Subject keywords would certainly be useful. Since one document will generally have many associated keywords, we would introduce another tupleset, say DocKeyword, to represent the new information. Should keywords be entities? Again there is a tradeoff, but the argument for entities seems persuasive: limiting the range of keywords increases the value of the database for retrieval. The keyword entities could also participate in relationships with a dictionary of synonyms, *Computing Reviews* categories, etc.

We should also consider multiple versions of a single document. If these relationships are to be represented in the database, we must at least factor the date and file fields of Document tuples out into tuples of a new type, say DocVersion. Other information associated with a version might be the identity of the version it was derived from, and a "change log" entry describing what changes were made.

This is certainly not a complete design, and the reader is encouraged to fit his or her own ideas into the database framework. A question to consider at the conclusion of your design exercise is: are you now willing to change the type of a document version's file attribute from STRING to CedarFileReference?

6. A sample program

As a simple example, we will consider the following concocted program, dealing with an imaginary database of people in a PersonDomain tupleset with name and age attributes, and a FriendRelation tupleset both of whose attributes are TupleRefs to tuples of the PersonDomain.

```
PersonDomain, FriendRelation: TupleSet;
pName, pAge: --PersonDomain-- Attribute;
fFriendee, fFriender: --FriendRelation-- Attribute;
harry, mark, rick, nori: -- PersonDomain -- Tuple;
```

The following routine defines the tuplesets and their attributes through calls to the database system. Since the version parameter to CreateTupleSet and CreateAttribute defaults to NewOnly, an ERROR would be raised if this program were executed a second time. The owner of the TupleSets here has been defaulted to no owner (i.e., public TupleSets).

```
Initialize: PROCEDURE =
  BEGIN
  indexOrder: ARRAY[0..1) OF Attribute ← [pAge];
  PersonDomain← CreateTupleSet[name: "Person"];
  pName← CreateAttribute[PersonDomain, "Name", string];
  pAge← CreateAttribute[PersonDomain, "Age", cardinal];
  []← CreateIndex[PersonDomain, DESCRIPTOR[indexOrder]];
  FriendRelation← CreateTupleSet[name: "Friend"];
  fFriendee← CreateAttribute[FriendRelation, "Friendee", tupleRef];
  fFriender← CreateAttribute[FriendRelation, "Friender", tupleRef];
  END;
```

The following routine creates various tuples in the tuplesets. Several methods are used to assign the tuple attributes to illustrate the alternatives.

```
InsertData: PROCEDURE =
  BEGIN t: Tuple;
  myName: STRING ← "Rick Cattell";
  myAge: CARDINAL ← 26;
  avl: ARRAY[0..2) OF AttributeValue←
    [[pName, S2V["Nori Suzuki"]], [pAge, C2V[30]]];
  SetFT: PROCEDURE[t: Tuple, a: Attribute, v: Tuple] =
    INLINE BEGIN SetF[t, a, T2V[v]] END;
  harry← CreateTuple[PersonDomain];
  mark← CreateTuple[PersonDomain];
  rick← CreateTuple[PersonDomain];
  nori← CreateTuple[PersonDomain];
  SetFS[harry, pName, "Harry Q. Bovik"];
  SetFS[harry, pAge, "32"];
  SetF[mark, pName, S2V["Mark Brown"]];
  SetF[mark, pAge, C2V[27]];
  SetF[rick, pName, S2V[myName]];
  SetF[rick, pAge, C2V[myAge]];
  SetMultipleF[ nori, DESCRIPTOR[avl] ];
  t← CreateTuple[FriendRelation];
  SetFT[t, fFriendee, rick]; SetFT[t, fFriender, harry];
  t← CreateTuple[FriendRelation];
```

```
  SetFT[t, fFriendee, nori]; SetFT[t, fFriender, harry];
  END;
```

The following routine prints all of the people in the PersonDomain, by doing a ForMatchSet retrieval with no constraints. Note that we have explicitly released tuples in this routine, and that a WriteStringD routine which deallocates its string argument is used on string retrieved from fields; these calls will not be necessary when the garbage collector is used.

```
PrintEveryone: PROCEDURE =
  BEGIN
  person: -- PersonDomain -- Tuple;
  ms: MatchSet;
  WriteChar[CR];
  WriteLine["PersonDomain:"]; WriteChar[CR];
  WriteLine["Name                    Age"];
  ms← ForMatchSet[PersonDomain];
  UNTIL Null[person← NextMatch[ms]] DO
    WriteStringD[GetFS[person, pName]]; WriteString["        "];
    WriteDecimal[V2C[GetF[person, pAge]]]; WriteChar[CR];
    ReleaseTuple[person];
    ENDLOOP;
  ReleaseMatchSet[ms];
  END;
```

The following routine prints some of the people in the PersonDomain, by doing a ForMatchSet retrieval requesting the tuples with the age attribute in the range 30 to 100.

```
PrintSomeOnes: PROCEDURE =
  BEGIN
  person: -- PersonDomain -- Tuple;
  ms: MatchSet;
  oldGuys: ARRAY [0..1) OF AttributeMatch←
    [[attribute: pAge, lowValue: C2V[30], highValue: C2V[100]]];
  WriteChar[CR];
  WriteLine["People between 30 and 100:"];  WriteChar[CR];
  WriteLine["Name                    Age"];
  ms← ForMatchSet[PersonDomain, DESCRIPTOR[oldGuys]];
  UNTIL Null[person← NextMatch[ms]] DO
    WriteStringD[GetFS[person, pName]]; WriteString["        "];
    WriteStringD[GetFS[person, pAge]]; WriteChar[CR];
    ReleaseTuple[person];
    ENDLOOP;
  ReleaseMatchSet[ms];
  END;
```

The following routine prints all of the tuples in the PersonDomain that are related through the FriendRelation to the tuple in the PersonDomain named harry.

```
PrintFriends: PROCEDURE =
  BEGIN
  friendRelationship: --FriendRelation-- Tuple;
  friend: --PersonDomain-- Tuple;
```

```
  rs: RefSet;
  WriteChar[CR]; WriteLine["Harry's friends:"]; WriteChar[CR];
  rs← ForRefSet[harry, fFriender];
  UNTIL Null[friendRelationship← NextRef[rs]] DO
    friend← V2T[GetF[friendRelationship, fFriendee]];
    WriteStringD[GetFS[friend, pName]];
    WriteString["          "]; WriteChar[CR];
    ReleaseTuple[friend]; ReleaseTuple[friendRelationship];
    ENDLOOP;
  ReleaseRefSet[rs];
  END;
```

The main program opens the default database, calls the above routines, and calls CloseDatabase to cause the updates to be made. There are now 8 new dictionary tuples and 6 new data tuples in the database.

```
[]← OpenDatabase[];
Initialize[];
InsertData[];
PrintEveryone[];
PrintSomeOnes[];
PrintFriends[];
CloseDatabase[];
```

7. References

[Cattell] R.G.G. Cattell, "An entity-based database interface", CSL-79-9, August 1979.

[Date] C.J. Date, *An Introduction to Database Systems*, Addison-Wesley, 1977.

[Hoare] C.A.R. Hoare, "Data reliability", *Proc. International Conference on Reliable Software*, Los Angeles, April 1975, p 528-533.

[DFS] Jay E. Israel, James G. Mitchell, and Howard E. Sturgis, "Separating data from function in a distributed file system", CSL-78-5, September 1978.

CSL Notebook Entry

To:	Alpine designers	Date:	February 20, 1981
From:	Mark Brown	Location:	PARC/CSL
Subject:	FileStore transaction code, version 1	File:	[Ivy]<Alpine>Doc>FileStoreTransCode1.bravo

XEROX

Attributes: informal, technical, Database, Distributed computing, Filing

References: [Ivy]<Alpine>Doc>FileStoreDesign1.bravo

Abstract: This memo contains Mesa-like code implementing the Transaction Coordinator and Transaction Worker sections of FileStore.

The following paragraphs provide some context for the code. The code is organized around the manipulation of volatile objects: collectible records, monitors, etc. Since this is a concurrent program there is no choice; we don't have stable collectible records or stable monitors. But the crucial invariants of the program involve the contents of the disk, since this is all that is available during crash recovery.

The log

The *log* is an infinite, sequential write-once, random access read, store for *log records.* The log is implemented using a circular buffer of disk pages. We make it seem infinite by (1) writing a page on tape before reusing it, and (2) never permitting a transaction to run so long that it would require access to portions of the log that now reside on tape. We achieve (2) by aborting any long-running transaction. The tape copy of the log can be used for recovery from media failures. If this recovery is not needed, the tape copy is unnecessary.

Each log record is uniquely identified by its *log record ID.* Depending on the structure of log records or pages, this might be the index relative to the beginning of the log of the first (or last) byte of the log record. It should be possible to do a random access read to an online log record, given the log record ID.

We make *checkpoints* in the log in order to bound the amount of work that is required during recovery. A checkpoint record contains, among other things, a list of all active transactions at the time of the checkpoint (both transactions whose outcome is not yet determined, and transactions that are committed or aborted but still have work to do.)

The *restart record* is the "root" of the log. It points to the most recent checkpoint. The restart record should be written in two independent places in order to tolerate failures during write. The restart record itself should be found without searching.

At restart it is also necessary to find the end of the log (the most recently written log record) by reading forward from the most recent checkpoint. There seem to be several ways of structuring log records or log pages to achieve this.

Interface

We have not attempted to define a complete Mesa interface to the log. The code below uses the following three log calls.

Log.Write: PROC [record: LogRecord, forceLog: BOOLEAN] RETURNS [LogRecordID];
Writes *record* to the log, and returns its ID. If forceLog, then waits until *record* actually appears on the disk before returning.

Log.Force: PROC [] RETURNS [LogRecordID];
Returns the ID of the last record in the log. Waits until this record actually appears on the disk before returning.

Log.FindTransState: PROC [transID: TransID] RETURNS [{committed, aborted, unknown}];
Searches online log for outcome of given transaction.

The code below uses several log record types. The precise encoding should not be taken seriously; only the information carried by each record is significant. The encoding will have an impact on the complexity of the recovery algorithm, however.

```
Log.LogRecord: TYPE = RECORD [;
  transID: TransID,
  SELECT * FROM
    -- coordinator
    begin => [],
    registerWorker => [worker: FileStoreID],
    collecting => [workers: LIST OF FileStoreID, places: LIST OF FileStoreID],
    committing => [],
    aborting => [],
    committed => [],
    aborted => [],
    -- worker
    workerRegistered => [],
    workerReady => [places: LIST OF FileStoreID],
    workerCommitting => [],
    workerAborting => [],
    workerCommitted => [],
    workerAborted => [],
    -- both (one phase protocol on single FileStore)
    workerCommittingAndCoordinatorCommitted => [],
    workerAbortingAndCoordinatorAborted => [],
    -- ...
    ENDCASE ];
```

Recoverable states

We define the *recoverable state* of a transaction agent (coordinator or worker) to be the state to which it returns upon crash recovery. Only states recorded in logs are recoverable.

Recoverable states of a coordinator

Unknown. A transaction is *unknown* if (1) it has no COLLECTING record in the log, or (2) has either a COMMITTED or an ABORTED record in the log. (It has not entered the first phase of the commit process, or it has committed and carried out all of its intentions, or it has aborted and undone all of its pre-commit updates.)

Collecting. A transaction is *collecting* if (1) it has a COLLECTING record in the log, and (2) has no COMMITTING or ABORTING record in the log. (It wishes to commit, is collecting "ready votes" from workers, but has not yet decided the transaction outcome.)

Committing. A transaction is *committing* if it has a COMMITTING record in the log but has no COMMITTED record in the log. (It has committed but has not yet heard "ack" from all of its workers.)

Aborting. A transaction is *aborting* if it has a ABORTING record in the log but has no ABORTED record in the log. (It has aborted but has not yet heard "ack" from all of its workers.)

Note that the *outcome* of a transaction (*committed* or *aborted*) is not viewed as a recoverable state, but instead these two outcomes are merged into the *unknown* state. Recovery must be prepared to deal with transactions that are so old that no online record of them exists. These transactions might have committed or aborted, but our algorithms should be insensitive to this, since otherwise we would be forced to mount a tape and search for the old transaction.

We consider that part of the coordinator's role in the system is to respond to requests for information about the outcome of specific transactions (this is the motivation for encoding the coordinator's FileStore in a TransID.) Workers should have no part in this. As a practical matter, the system is unlikely to answer requests that cannot be resolved by searching the online log.

Recoverable states of a worker

Unknown. A transaction is *unknown* if (1) it has no READY record in the log, or (2) has either a COMMITTED or an ABORTED record in the log. That is, either it has not entered the first phase of the commit process, or it has committed and carried out all of its intentions, or it has aborted and undone all of its pre-commit updates.

Ready. A transaction is *ready* if (1) it has a READY record in the log, and (2) has no COMMITTING or ABORTING record in the log. That is, it has voted ready to the coordinator, but has not yet heard the outcome of the transaction from the coordinator.

Committing. A transaction is *committing* if it has a COMMITTING record in the log but has no COMMITTED record in the log. That is, it has committed but not carried out all of its intentions.

Aborting. A transaction is *aborting* if it has a ABORTING record in the log but has no ABORTED record in the log. That is, it has aborted but not undone all of its pre-commit updates.

Discussion

The worker states *committing* and *aborting* are an optimization. They allow less undo and redo work at restart by certifying that certain transactions are completely finished. This is also one of the functions of checkpoints, but checkpoints are taken less frequently than *committing* and *aborting* records are written. (Checkpoints themselves may be viewed as an optimization, since in principle the state of the file store can be restored by reading the log. In the same sense, the file store is an optimization over doing a linear scan of the log for each read access.)

Crash recovery

During crash recovery, we read the log to determine the recoverable state of each transaction that has been active since the most recent checkpoint. Then we perform the actions described below (later we'll have Mesa code for these too):

Coordinator

Unknown. If the transaction has no COMMITTED or ABORTED record in the log, we abort it (write ABORTED to the log, and optionally notify all of the workers whose identity has been logged); otherwise there is nothing to do.

Collecting. We call all workers again (they are identified in the COLLECTING record), asking for ready/notReady decisions.

Committing, Aborting. We call all workers again, telling them to commit or abort.

Worker

Unknown. If the transaction has no COMMITTED or ABORTED record in the log, we abort it (we *must* notify the coordinator); otherwise there is nothing to do.

Ready. We restore the transaction's volatile state from the log, reacquire the transaction's resources (locks), then wait for word from the coordinator.

Committing, Aborting. We restore the transaction's volatile state from the log. This mainly means reconstructing the buffer pool at the time of the crash, as well as lists of other deferred updates such as file shortening and deletion. Locks do not have to be reacquired for pages (unlike the ready case), since in normal operation the buffer pool is treated as "truth". We then set in progress whatever processes are normally used to carry out intentions or undo updates. These are carefully coded to be idempotent, so they can tolerate any number of crashes.

Remote procedure call

The remote calls in this code are FileStore[F].PrepareWorker, FileStore[F].FinishWorker, and FileStore[F].RegisterWorker, where F is a FileStoreID. I chose this strange syntax since it cannot possibly be confused with the syntax we'll end up with.

My assumption about a remote call is that when it returns, it has executed at the remote site one or more times. The result of the call, if any, is the result of one of the calls at the remote site.

I do not quite understand what happens when the remote call fails because the remote site doesn't respond and we time out. I would prefer *not* to catch a signal in this case. But this leaves out procedures that return no results.

It may be helpful to consider the three remote procedures used here. If a call to RegisterWorker fails, then the proper response is to abort the operation that required the RegisterWorker. At a higher level, the client may then decide to abort the transaction. If a call to PrepareWorker fails, we must assume the worst: a call got through and the worker is, in fact, ready. The coordinator can decide to abort the transaction, but he retains the responsibility to notify the worker of this fact. Similarly, if a call to FinishWorker fails, we must repeat the call until it succeeds, since the worker must be assumed ready.

Deficiencies

There is some potential for useful special-casing of readonly workers. They do not have to participate in phase two of commit, since commit and abort are the same for a readonly worker. The code does not take advantage of this.

The implementation of FinishTransaction is crude since it does not attempt to contact workers in parallel. In a real implemetation, we expect to have a group of processes whose role in life is to call PrepareWorker and FinishWorker. The process doing the FinishTransaction might be responsible for one of the calls, or (more likely) none of them. In the latter case, it could wait on a CV associated with the Coordinator record, and be notified as the responses arrived.

Recovery needs to be coded to a similar level of detail. A strategy for cleaning up obsolete Coordinators and Workers needs to be specified; the current thing would accumulate them indefinitely.

```
CoordinatorImpl: MONITOR LOCKS t USING t: Coordinator
  IMPORTS Log, CoordinatorTable EXPORTS FileStore = BEGIN

CreateTransaction: PUBLIC PROC [stableCreation: BOOLEAN] RETURNS [t: TransID] = {
  c: Coordinator ← ConsCoordinator[
    transID: [
      coordinatorFileStoreID: FileStorePrivate.MyFileStoreID[],
      idOnFileStore: FileStorePrivate.UniqueLongCardinalOnMyFileStore[]],
    transState: unknown];
  c.beginRecordID ← Log.Write[record: [c.transID, begin[]], forceLog: stableCreation];
  c.state ← active; IF stableCreation THEN c.recoverableState ← active;
  RETURN [c.transID] };

ConsCoordinator: PROC [transID: TransID, transState: TransState] RETURNS [c: Coordinator] = {
  c ← NEW[CoordinatorObject ← [
    transID: transID, state: transState, recoverableState: transState,
    coordinator: MyFileStoreID[] ]];
  CoordinatorTable.Insert[c];
  RETURN[c] };

CoordinatorFileStoreIDFromTransID: PUBLIC PROC [t: TransID] RETURNS [FileStoreID] = {
  RETURN [t.coordinatorFileStoreID] };

RegisterWorker: PUBLIC ENTRY PROC [t: TransID, w: FileStoreID, stableRegistration: BOOLEAN]
  RETURNS [success: BOOLEAN] = {
  -- The caller (a Worker) should have no monitors locked.
  c: Coordinator ← CoordinatorFromTransID[t: t, whereToLook: volatileOnly];
  IF c = NIL OR c.state # active THEN RETURN [FALSE];
  IF ~IsMember[w, c.workers] THEN {
    c.workers ← Append[w, c.workers];
    [] ← Log.Write[record: [t, registerWorker[w]], forceLog: stableRegistration] };
  RETURN [TRUE] };

CoordinatorFromTransID: PROC [t: TransID, whereToLook: {volatileOnly, volatileOrLog}] RETURNS
  [Coordinator] = {
  IF CoordinatorFileStoreIDFromTransID[t] # FileStorePrivate.MyFileStoreID[] THEN ERROR;
  c: Coordinator ← CoordinatorTable.Lookup[t];
  IF c # NIL OR whereToLook = volatileOnly THEN RETURN[c];
  -- No volatile record of t exists. t has been recorded in the log unless it was created with stableCreation = FALSE. Search the
    online portion of the log for the outcome of t. (We should keep volatile records of recent transactions to keep the frequency
    of this operation down to an acceptable level. By doing some work at checkpoint time, only checkpoint records need to be
    looked at, and these are linked together. We can use the fact that TransIDs are generated in a fixed order to bound the
    search in the case that no record was made of a transaction.)
  s: TransState ← Log.FindTransState[t];
  SELECT s FROM
    committed, aborted => NULL;
      -- normal case.
    unknown => s ← FileStorePrivate.ResolveUnknownTransaction[t];
      -- t not found in log; appeal to a higher authority.
    active, collecting, committing, aborting => ERROR;
    ENDCASE => ERROR;
  RETURN[ConsCoordinator[t, s]] };
```

```
FinishTransaction: PUBLIC ENTRY PROC [t: TransID, requestedOutcome: {commit, abort},
  commitRecordPlaces: LIST OF FileStoreID]
  RETURNS [outcome: {committed, aborted, unknown}] = {
  -- To notify its coordinator of a spontaneous abort, a worker causes FinishTransaction[t, abort, NIL] (with no Worker monitors
    locked).
  c: Coordinator ← CoordinatorFromTransID[t: t, whereToLook: volatileOrLog];
  IF c = NIL THEN RETURN [unknown];
  DO SELECT c.state FROM
    active => EXIT;
      -- Normal case.
    committed => RETURN [committed];
    aborted => RETURN [aborted];
    collecting => WAIT c.outcomeDetermined;
      -- Another activation of this procedure is at work. Wait for it to finish.
    ENDCASE => ERROR; -- includes committing, aborting, unknown.
    ENDLOOP;
  IF c.workers = NIL THEN {
    -- Funny transaction; no workers! Don't complain.
    [] ← Log.Write[record: [t, requestedOutcome-ed[]], forceLog: TRUE];
    c.recoverableState ← outcome ← requestedOutcome; GOTO done };
  IF commitRecordPlaces = NIL THEN commitRecordPlaces ← LIST [c.coordinator];
  IF c.workers.first = c.coordinator AND c.workers.rest = NIL AND
    commitRecordPlaces.first = c.coordinator AND commitRecordPlaces.rest = NIL THEN {
    -- Single FileStore case; use one phase protocol. Worker will do the necessary log write.
    c.recoverableState ← outcome ← FileStorePrivate.PrepareAndFinishWorker[t, requestedOutcome];
    GOTO done };
  IF requestedOutcome = abort THEN {
    -- Abort request. Use one phase protocol.
    [] ← Log.Write[record: [t, aborting[]], forceLog: TRUE];
    FOR w: FileStoreID IN c.workers DO
      [] ← FileStore[w].PrepareWorker[transID: t, requestedOutcome: abort,
        commitRecordPlaces: NIL];
      ENDLOOP;
    [] ← Log.Write[record: [t, aborted[]], forceLog: FALSE];
    c.recoverableState ← outcome ← aborted; GOTO done };
  -- Trying to commit a multi FileStore transaction. Use two phase protocol.
  [] ← Log.Write[record: [t, collecting[c.workers, commitRecordPlaces]], forceLog: TRUE];
  c.state ← c.recoverableState ← collecting;
  -- We are now responsible for doing the job. To do this properly we should have a number of processes whose role in life is to
    call PrepareWorker or FinishWorker, and wait for the return. If we wait in the monitor, others who enter will wait for us to
    determine the outcome. In this crude implementation, we do not wait.
  outcome ← committed;
  FOR w: FileStoreID IN c.workers DO
    IF FileStore[w].PrepareWorker[transID: c.transID, requestedOutcome: requestedOutcome,
      commitRecordPlaces: NIL] = notReady THEN outcome ← aborted;
    ENDLOOP;
  [] ← Log.Write[record: [t, outcome-ing[]], forceLog: TRUE];
  c.state ← c.recoverableState ← outcome-ing;
  FOR w: FileStoreID IN c.workers DO
    FileStore[w].FinishWorker[transID: c.transID, requestedOutcome: outcome];
    ENDLOOP;
  [] ← Log.Write[record: [t, outcome-ed[]], forceLog: FALSE];
  c.recoverableState ← outcome-ed; GOTO done;
  EXITS
    done => RETURN[c.state ← outcome]
  };
END.--CoordinatorImpl
```

```
WorkerImpl: MONITOR LOCKS w USING w: Worker
  IMPORTS Log, WorkerPrivate, WorkerTable EXPORTS FileStore, FileStorePrivate = BEGIN
PrepareWorker: PUBLIC PROC [t: TransID, requestedOutcome: {commit, abort},
  commitRecordPlaces: LIST OF FileStoreID]
  RETURNS [{readonlyReady, ready, notReady}] = {
  PrepareWorkerEntry: ENTRY PROC [w: Worker, requestedOutcome: {commit, abort},
    commitRecordPlaces: LIST OF FileStoreID]
    RETURNS [{readonlyReady, ready, notReady}] = {
    SELECT w.state FROM
      active => NULL; -- normal case.
      ready => GOTO readyReturn;
      committed => RETURN [ready]; --result is arbitrary
      aborted => RETURN [notReady]; --result is arbitrary
      ENDCASE => ERROR;
    IF requestedOutcome = commit AND PrepareToCommit[w: w, twoPhase: TRUE] THEN {
      [] ← Log.Write[record: [w.transID, workerReady[commitRecordPlaces]], forceLog: TRUE];
      w.state ← w.recoverableState ← ready;
      GOTO readyReturn }
    ELSE {
      AbortWorker[w]; RETURN [notReady] }
    EXITS
      readyReturn => RETURN [IF w.readonly THEN readonlyReady ELSE ready];
    };--PrepareWorkerEntry
  w: Worker ← WorkerTable.Lookup[t];
  IF w = NIL THEN RETURN [notReady];
  RETURN [PrepareWorkerEntry[w, requestedOutcome, commitRecordPlaces]];
  };

FinishWorker: PUBLIC PROC [t: TransID, requiredOutcome: {commit, abort}] = {
  PrepareWorkerEntry: ENTRY PROC [w: Worker, requiredOutcome: {commit, abort}] = {
    SELECT w.state FROM
      ready => NULL; -- normal case.
      ENDCASE => RETURN; --we must have already finished.
    IF requestedOutcome = commit THEN CommitWorker[w: w, twoPhase: TRUE]
    ELSE AbortWorker[w: w, twoPhase: TRUE] };
  w: Worker ← WorkerTable.Lookup[t];
  IF w = NIL THEN RETURN; --we must have already finished.
  FinishWorkerEntry[w, requiredOutcome] };

PrepareAndFinishWorker: PUBLIC PROC [t: TransID, requestedOutcome: {commit, abort}]
  RETURNS [{committed, aborted}] = {
  PrepareAndFinishWorkerEntry: ENTRY PROC [w: Worker, requestedOutcome: {commit, abort}]
    RETURNS [{committed, aborted}] = {
    SELECT w.state FROM
      active => NULL; -- normal case.
      ENDCASE => ERROR; --this is local only, and protected by coordinator's monitor.
    IF requestedOutcome = commit AND PrepareToCommit[w: w, twoPhase: FALSE] THEN {
      CommitWorker[w: w, twoPhase: FALSE]; RETURN [committed] }
    ELSE {
      AbortWorker[w: w, twoPhase: FALSE]; RETURN [aborted] }
  w: Worker ← WorkerFromTransID[t: t, createWorkerIfNecessary: FALSE];
  IF w = NIL THEN ERROR; --this is local only, and protected by coordinator's monitor.
  PrepareAndFinishWorkerEntry[w, requestedOutcome] };

PrepareToCommit: PROC [w: Worker, twoPhase: BOOLEAN] RETURNS [ready: BOOLEAN] = {
  -- Here we must check to ensure that the transaction has all of the resources necessary for commitment. This is much easier if
    they have actually been allocated already, for instance file creation and file lengthening not deferred. If updates have been
    deferred using update mode locks, these must be converted to exclusive mode at this time. If twoPhase = TRUE, then all
    exclusive locks must be written into the log.
  RETURN [ ... ] };
```

```
CommitWorker: PROC [w: Worker, twoPhase: BOOLEAN ← TRUE] = {
  IF twoPhase THEN
    [] ← Log.Write[record: [w.transID, workerCommitting[]], forceLog: TRUE];
  ELSE
    [] ← Log.Write[record: [w.transID, workerCommittingAndCoordinatorCommitted[]],
      forceLog: TRUE];
  -- Here we perform the volatile actions associated with transaction commitment. These are: make all buffer pool entries written
    by this transaction be the "truth" versions (discard old versions if present in buffer pool). Carry out other deferred updates
    such as file deletion and shortening. Finally, release locks held by this transaction.
    [] ← Log.Write[record: [w.transID, workerCommitted[]], forceLog: FALSE];
  };

AbortWorker: PROC [w: Worker, twoPhase: BOOLEAN ← TRUE] = {
  IF twoPhase THEN
    [] ← Log.Write[record: [w.transID, workerAborting[]], forceLog: FALSE];
  ELSE
    [] ← Log.Write[record: [w.transID, workerAbortingAndCoordinatorAborted[]],
      forceLog: FALSE];
      -- Log force is not required since transaction will be aborted in recovery (but we might do it anyway if it turns out to
        simplify recovery).
  -- Here we perform the volatile actions associated with transaction abortion. These are: discard all buffer pool entries written
    by this transaction. Undo non-deferred updates such as file creation and lengthening. Finally, release locks held by this
    transaction.
    [] ← Log.Write[record: [w.transID, workerAbortted[]], forceLog: FALSE];
  };
END.--WorkerImpl
```

```
WorkerImpl2: MONITOR
  IMPORTS Log, WorkerTable EXPORTS FileStorePrivate = BEGIN

CreateWorker: PUBLIC PROC [t : TransID, stableCreation: BOOLEAN] RETURNS [Worker] = {
  -- This implementation ensures that (1) no duplicate Workers (two Workers with the same transID) are created, (2) no monitors
    are locked when a remote call to RegisterWorker is made. This implementation sometimes makes a redundant remote call
    to RegisterWorker (in the case that two processes try to migrate the same transaction to the same FileStore at about the same
    time), but RegisterWorker must be prepared for duplicated calls anyway.
  LogWorker: ENTRY PROC [w: Worker] = {
    IF w.state = unknown THEN {
      [] ← Log.Write[record: [t, workerRegistered[]], forceLog: FALSE]; w.state ← active };
    IF stableCreation AND w.recoverableState = unknown THEN {
      [] ← Log.Force[]; w.recoverableState ← active } };
  w: Worker ← WorkerTable.Lookup[t];
  IF w # NIL THEN RETURN[w];
  IF FileStore[CoordinatorFileStoreIDFromTransID[t]].RegisterWorker[
    t, MyFileStoreID[], stableCreation] THEN {
    w ← ConsWorker[transID: t, transState: unknown];
    LogWorker[w] };
  RETURN [w] };

ConsWorker: ENTRY PROC [transID: TransID, transState: TransState] RETURNS [w: Worker] = {
  -- This procedure never takes very long, which is good since every transaction creation passes through this monitor.
  IF (w ← WorkerTable.Lookup[t]) # NIL THEN RETURN[w];
  w ← NEW[WorkerObject ← [
    transID: transID, state: transState, recoverableState: transState,
    coordinator: CoordinatorFileStoreIDFromTransID[transID],
    worker: FileStorePrivate.MyFileStoreID[] ]];
  WorkerTable.Insert[w];
  RETURN[w] };
END.--WorkerImpl2
```

```
FileStorePrivate: DEFS = BEGIN

FileStoreID: TYPE = MACHINE DEPENDENT RECORD [
  ...];
FileObject: TYPE = RECORD [
  file: FileID,
  trans: Worker,
  fileLockMode: LockMode];

TransactionID: TYPE = MACHINE DEPENDENT RECORD [
  coordinatorFileStoreID: FileStoreID,
  idOnFileSystem: LONG CARDINAL];
  -- Note that a FileStoreID, not a MachineID, is needed (because of
     CoordinatorFileStoreIDFromTransID).
TransState: TYPE = MACHINE DEPENDENT
  {unknown, active, ready, collecting, committing, aborting, committed, aborted};
  -- or something like that ... this is the union of coordinator and worker states.
CoordinatorObject: TYPE = MONITORED RECORD [
  transID: TransID,
  state, recoverableState: TransState,
  beginRecordID: LogRecordID ← NULL,
  coordinator: FileStoreID,
  workers: LIST OF FileStoreID ← NIL];
WorkerObject: TYPE = MONITORED RECORD [
  transID: TransID,
  state, recoverableState: TransState,
  coordinator: FileStoreID,
  worker: FileStoreID];


MyFileStoreID: PROC [] RETURNS [FileStoreID];

UniqueLongCardinalOnMyFileStore: PROC [] RETURNS [LONG CARDINAL];

ResolveUnknownTransaction: PROC [t: TransID] RETURNS [outcome: TransState {committed,
  aborted}];

PrepareAndFinishWorker: PROC [t: TransID, requestedOutcome: {commit, abort}]
  RETURNS [{committed, aborted}];
  -- Caller (an implementation of BFS.FinishTranasction) asserts that w is the only worker of the
     transaction and that w.worker = w.coordinator. Caller requests w to finish with the specified
     outcome, writing the COMMIT record on w.coordinator. Worker can return the opposite of the
     requested outcome.

CreateWorker: PROC [t: TransID] RETURNS [Worker];
  -- If worker exists for this transaction, just returns it; otherwise creates it.

END.--FileStorePrivate


CoordinatorTable: DEFS = BEGIN
  -- This is a monitor.

Insert: PROC [c: Coordinator];
  -- Caller asserts that no Coordinator x in the table has x.transID = c.transID. Add c to the table.

Lookup: PROC [t: TransID] RETURNS [Coordinator];
  -- Return c in table such that c.transID = t; return NIL if no such c exists.

Delete: PROC [c: Coordinator];
  -- Caller asserts that c is in the table. Remove c from table.
```

END.--CoordinatorTable

```
WorkerTable: DEFS = BEGIN
  -- This is a monitor.
Insert: PROC [w: Worker];
  -- Caller asserts that no Worker x in the table has x.transID = w.transID.  Add w to the table.
Lookup: PROC [t: TransID] RETURNS [Worker];
  -- Return w in table such that w.transID = t; return NIL if no such w exists.
Delete: PROC [w: Worker];
  -- Caller asserts that w is in the table.  Remove w from table.
END.--WorkerTable
```

CSL Notebook Entry

To:	Alpine designers	Date:	February 24, 1981
From:	Mark Brown	Location:	PARC/CSL
Subject:	Alpine lock manager concepts, version 0	File:	[Ivy]<Alpine>Doc>LockConcepts0.bravo

XEROX

Attributes: informal, technical, Database, Distributed computing, Filing

Abstract: This memo proposes the scope of the Alpine lock manager. It is largely an interpretation and restatement of several papers on the IMS and System R lock managers. It does *not* propose an interface; see LockDesign*.bravo.

Papers on locking

I have a copy of each paper listed here. This memo should make it unnecessary for everyone to read all of these papers, but I have found them useful. I have read some of them much more carefully than others.

J. N. Gray, R. A. Lorie, G. R. Putzolu, I. L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," in G. M. Nijssen (ed) *Modelling in Data Base Management Systems*, North Holland, 1976. (Describes the System R lock manager.)

J. N. Gray, "Notes on Data Base Operating Systems," IBM San Jose Research Report RJ2188, February 1978, Section 5.7. Also appeared in *Operating Systems - an Advanced Course*, Springer Lecture Notes in Computer Science 60, 1978. (There is a lot of duplication between this and the preceding paper.)

J. N. Gray, "Experience with the System R lock manager," slides for an oral presentation given in Berkeley, 1980. (Describes the frequency of wait and deadlock, endorses update mode locks.)

H. F. Korth, "Locking protocols: general lock classes and deadlock freedom," draft of Princeton University Ph.D. dissertation, 1981. (A clean treatment of a System R style lock manager. Describes how to derive compatability properties of locks systematically, and gives a variable-granularity, deadlock-free locking protocol. But this protocol only prevents deadlock due to lock contention; see below.)

R. Obermarck, "IMS/VS program isolation feature," IBM San Jose Research Report RJ2879, July 1980. (Describes the IMS lock manager.)

R. Obermarck, "Global deadlock detection algorithm," IBM San Jose Research Report RJ2845, June 1980. (Describes the System R* distributed deadlock detection algorithm.)

R. Obermarck, "Deadlock detection for all resource classes," IBM San Jose Research Report RJ2955, October 1980. (Points out that deadlocks occur due to contention for resources that aren't covered by the usual sort of lock: network connections and heap storage are examples. Then gives an algorithm for detecting deadlocks over all these resources.)

A top-down view of locks

Suppose that a program P performs a sequence of operations on a collection of files F. These files satisfy some application-specific consistency constraints. P is written so that if F satisifes the consistency constraints, then running P will bring F to another state that satisifes the consistency constraints.

Now consider that several programs may access the file system simultaneously. Individual operations on files (read page, write page) can be made atomic by using monitors (for instance, a monitor can keep one program from seeing a partially completed page write performed by another program.) *Transactions* are used to make *sequences* of operations atomic, so that programs P and Q can execute simultaneously and yet each can preserve its consistency constraints. This is done by ordering the operations of different transactions so that their execution equivalent to some separate execution of the transactions (P before Q, or P after Q.) When such as ordering is imposed we say that the *schedules* of the two transactions are *serializable.* It is not necessary to actually run one transaction before the other: if P and Q both read a page and neither updates it, then both reads may complete before either transaction finishes.

Locking is one technique for serializing the schedules of concurrent transactions. Consider the example of reading and writing a page. Suppose that we impose these rules on transactions that wish to read or write: (1) Before reading a page, obtain a read lock on it; before writing obtain a write lock. (2) At no time may a write lock coexist with any other lock (read or write) on a page, but a transaction may *convert* its read lock to a write lock without releasing the read lock. (3) No transaction may release any lock before it commits or aborts; after committing or aborting it must release all its locks. It can be proved that these rules allow only serializable schedules.

While the example just given is simple, it contains most of the features of more complicated locking protocols that are used in practice. Rule (1) is typical; we must lock an entity before using it. Locking protocols that allow locking of the same entity at different granularities (locking a page implicitly by locking the file containing it) have more elaborate versions of rule (1). Rule (2) governs the *compatability* of different lock requests for the same entity; the exact compatability rule depends on the semantics of the operations that are allowed by each type of lock. Rule (3) is called the *two phase* requirement: a transaction consists of two phases, a growing phase in which locks are acquired and none are released, and a shrinking phase in which locks are released and none are acquired. The two phase requirement seems to be part of all locking protocols that produce serializable schedules.

A bottom-up view of locks

Locks are used to maintain the consistency of shared data under concurrent access. The *lock manager* (i.e. the module that implements locks) is a low-level scheduler. It decides who must wait, but not who should run among the runnable processes. Other system components provide load control and other processor scheduling functions.

The basic lock call is "lock entity E in mode M for use by transaction T". An example of an entity is a file page; an example of a mode is "read" or "write". We say that the request is *granted* if the above call returns with the lock set. A program holding a lock in a certain mode has *permission* to operate on the locked entity in a way specified by the mode. Not all requests are granted right away; the caller may have to wait for another transaction to release a lock. Whether a lock request is be granted may be a function of the request's mode M, the modes of existing locks on entity E (for transaction T and for other transactions), and the modes of requests already made but not yet granted for locks on entity E.

To lock an entity we must name it. From the lock manager's point of view, the simplest form of name is a fixed-length string of bits. The lock manager is not concerned with the mapping between names and entities, and knows little or nothing about the structure of names. This allows new clients to use the lock manager without changing the lock manager or its interface.

Monitors are used to implement locks. Data structures whose sharing is controlled by locks need not be protected by monitors. (So in terms of the previous top-down discussion, locks may be used both for guaranteeing the atomicity of whole transactions and for providing the atomicity of their individual operations.) The primary differences between locks and monitors are (1) monitors only allow exclusive access to the monitored structure, while locks allow other policies such at multiple readers, (2) a monitor deadlock is a fatal error, while a lock deadlock can be resolved by rolling back one of the transactions involved.

Because locks are not integrated into the language as monitors are, locks require extra care on the part of the programmer. Clients who are programmed in terms of locks must obey a *lock protocol*: they must lock the entities that they intend to access before accessing them. The lock manager only ensures that if all its clients obey the lock protocol, then they will not interfere with each other in unexpected ways. (The same is really true about monitors, but the language makes it convenient to follow the correct monitor protocol and the compiler catches certain blunders.) The lock manager is also responsible for detecting deadlocks, and for breaking them by asking the recovery manager to roll back a transaction (releasing locks in the process.) (Note that it is the locks themselves that make backup practical by isolating transactions from one another and hence avoiding cascading backups.)

There are many natural forces that push complexity into the lock manager. We add logically redundant lock modes, such as update mode, to describe our application more precisely and allow higher concurrency and fewer deadlocks. We add a variable-granularity lock protocol (which from the lock manager's point of view reduces to the intention lock modes) to reduce the number of calls on the lock manager, since lock calls can amount to a significant fraction of the time in a system. Fortunately, adding a new lock mode has no impact on the complexity of the lock manager.

The exclusive locks set by an application typically parallel its "intentions" to perform writes during phase two of commit; hence it is tempting to widen the lock manager interface and thereby avoid a duplication of effort. We may also complicate the lock manager by giving it advice on scheduling decisions, for instance who to abort first in a deadlock situation.

An added complexity in System R is *lock escalation.* When a transaction acquires a certain number of locks, System R attempts to trade the most frequently-occurring locks (e.g. page locks on a particular file) for a single lock that covers them all. This trades concurrency for space, but it has worked well, and the code to implement escalation is straightforward. It does add some complexity and may widen the interface.

An added complexity in the Juniper system is *range locks.* In Juniper, all locks are considered to cover a range of entities. This allows locks to be set on ranges of bytes in a file. Range locks require that the lock data structure represent the linear ordering among certain locks; this means that at some point a list or tree representation must be used instead of simple hashing.

Lock modes and conversions

Locking an entity in a specific mode grants the holder of the lock permission to perform certain operations on the entity. By considering the potential interactions among these operations, we can derive the lock compatability predicate

```
Compat: PROC [request: LockMode, existing: LockMode] RETURNS [compatible: BOOLEAN].
```

If Compat[x, y], then a request for a lock of mode x can be granted even when there are existing locks of mode y for other transactions. It is clear that Compat[Read, Write] = FALSE is necessary, and that Compat[Read, Read] = TRUE is ok (we could make it FALSE too, but that would reduce concurrency.)

A *conversion* is a request by a transaction T to lock an entity E that is already locked by T. If T locks E in mode y, and there are no other locks on E, then clearly we should allow T to lock E in mode x even if ~Compat[x, y]. This explains why we treat conversions as being different from

other requests. In System R, conversions are always granted immediately if possible, even if other requests are waiting. (This is a violation of fair scheduling, but favoring a transaction T that already holds a lock is probably a good idea: granting other requests can only make it harder for T to proceed and eventually release the lock it has.)

Conversions are governed by a supremum function

Sup: PROC [request: LockMode, existing: LockMode] RETURNS [LockMode].

To honor a request for mode x when the existing mode is y, the lock manager converts the existing lock to mode Sup[x, y].

Compat and Sup contain all of the mode-specific knowledge in the lock manager.

Variable-granularity locking

A program that performs a sequential scan of a file wants to lock the whole file before it starts, thereby saving the space and time required for one lock per page (or one lock per tuple.) A program that does random-access into a file via an index structure wants to lock only the data that it depends on, so that it can run concurrently with other programs doing the same thing. This is the motivation for locking at different granularities.

Variable-granularity locking requires a more elaborate locking rule than "lock an entity before using it," since an entity might be locked implicitly by a lock on a larger entity that contains it. Therefore to lock, say, a page of a file, we first lock the file in an *intention mode*, then lock the page. The compatability predicate values for the intention modes ensure that consistency is maintained. As a result, the lock manager "knows" nothing about variable-granularity locking; it just increases the number of lock modes a bit.

Sets of locks

Clients of the lock manager may wish it to implement multiple sets of locks for a given transaction. IMS, System R, and the DBTG data manipulation language do this. The applications seem to be:

(1) Even when transactions use two phase locking, it is sometimes useful to obtain some locks for a short period and then release them. (I am not sure why; it may be that they use locks where we would use monitors.) To avoid double bookkeeping, the lock manager should maintain the locks as a set and release them with a single call.

(2) System R supports a notion of *save point* for a transaction. This is a checkpoint, i.e. a point to which a transaction can roll back in case of lock conflict. If the lock manager knows about sets of locks, its deadlock detection can pick a set of locks to abort rather than a transaction. If this set corresponds to a save point, the transaction can back up just that far.

In any case, it seems essential to have a lock manager call that releases all locks held by a given transaction. Another call that may be useful is one that enumerates a transaction's locks and allows the caller to downgrade them (from write to read); this is what happens during a Juniper "checkpoint" (commit while preserving the transaction and holding locks.) So a set structure will be maintained on a per-transaction basis. It is may not be much extra effort to allow multiple sets per transaction.

Scheduling competing requests

The System R lock manager uses the following scheduling strategy. If there are no waiters from other transactions for a *new* request (lock not already held in any mode by the transaction) and it can be granted immediately, then it is granted; otherwise the request is placed at the end of the

queue of waiters for this lock. If the request is a *conversion* (the lock is already held in some mode by the transaction) and it can be granted immediately, then it is granted (regardless of whether anyone is waiting); otherwise, the request is placed after any waiting conversions in the queue, but before any waiting non-conversions. (Note that a waiter can change from being a non-conversion to being a conversion while it is waiting, if another request from the same transaction is granted. In general, waiting requests from a single transaction are kept consecutive in the queue, and are ordered not by insertion time but by strength, with the weakest mode first).

Deadlock detection

The System R lock manager performs periodic deadlock detection by constructing the waiting-for graph and looking for cycles. (T1 is waiting for T2 if T1 is waiting for a lock on E in mode x, T2 holds a lock on E in mode y, and ~Compat[x, y].) It constructs the graph only if there are two or more waiters in the system. The graph is constructed from scratch each time, rather than being updated incrementally, since we expect deadlock to be rare even when compared with waiting (which is rare compared with locking.) Once a cycle is found, a victim is selected and aborted; in the absence of other information, it makes sense to select the cheapest victim. System R estimates the "cost" of a transaction to be the number of bytes of log it has written.

The paper by Obermarck discusses the elaborations on this that are used by System R* in a distributed, slow-communications environment. We may want to do something different in a distributed, fast RPC environment.

Timeouts must be used to abort long waiters, since the lock manager cannot detect all deadlocks. (A special problem in our environment is the detection of abandoned transactions, so they do not impede useful transactions for too long. This is not a simple problem in the Mesa world since it is hard to tell a crashed machine from a machine in the debugger. In Cedar the problem may be simpler. In any case, this is not a problem of the lock manager.)

Instrumentation

It seems quite useful to build an inexpensive statistics-gathering facility into the lock manager, and leave it on all the time. Knowing the frequency of wait and deadlock will be important in understanding overall system performance.

It would be nice to be able to communicate information on the time spent in lock wait back to individual clients (transactions) as well. Since the accuracy of this information does not need to be great, it should be possible to gather it cheaply.

Statistics on the System R lock manager

Time performance. In System R a lock/unlock pair takes 300 (IBM 370) instructions in the no-wait case, 5000 instructions in the wait case, 310 on the average (hence System R waits once per 500 calls to Lock). The CPU overhead for locking is typically 10% for transactions that do random access via B-Trees (fine-grained locks), and less than 1% for transactions that do mainly sequential processing (page or file locks).

Space performance. A System R lock occupies 40 bytes.

Program size. The System R lock manager is 3000 lines of PL/S, including the deadlock analyzer. System R was not conceived as a distributed system (though transactions involving multiple System R sites may be executed), and it does not detect multi-system deadlocks.

CSL Notebook Entry

To: Alpine designers

Date: February 25, 1981

From: Mark Brown

Location: PARC/CSL

Subject: Alpine lock manager design, version 0

File: [Ivy]<Alpine>Doc>LockDesign0.bravo

XEROX

Attributes: informal, technical, Database, Distributed computing, Filing

References: [Ivy]<Alpine>Doc>LockConcepts0.bravo

Abstract: This memo proposes an interface for the Alpine lock manager.

Introduction

This memo proposes a design for Alpine's Lock interface. Lock will be used by the implementation of FileStore, and also by applications that perform fine-grained locking and recovery.

Lock: DEFS = BEGIN

```
LockID: TYPE = RECORD [entity: System.UniversalID, subEntity: SubID];
SubID: TYPE [3];
  -- A file is a typical entity; a sub entity of a file is a page, the file's length, etc. If consecutive pages are
     represented as consecutive sub entities, then the lock manager may use a run-encoded
     representation. An application may use the entity field to encode what it thinks of as types:
     "database record", "index value", etc. The sub entity field must then be used to encode the name
     of a specific object of that type: a particular record or index value.
  -- Since lock mangers are responsible for locking disjoint sets of entities, there is no need for LockIDs
     to be unique across lock managers, much less across all space and time. They aren't unique IDs in
     the strongest sense.
LockMode: TYPE = { R, U, W, IR, IU, IW, RIU, RIW };
  -- or something like that .... R = read, U = update (read and probably convert to write), W = write,
     I = intention.

Lock: PROC [trans: TransID, lock: LockID, mode: LockMode];
  -- Acquire lock, in lock mode mode, for transaction trans. This means (1) wait until request can be
     granted, (2) acquire lock in mode, (3) increment count associated with [lock, trans]. The wait, if
     any, will be performed in the lock manager monitor, hence the client may not wish to call Lock
     with its own monitors locked; see LockNoWait below.
  -- <Our goal in designing this procedure should be to isolate the caller as much as possible from all of
     the things that can go wrong. Some method should be provided to communicate a long lock wait
     back to the caller - by an informational signal? return code? Will deadlock generate an ERROR?
     The case in which several calls from the same transaction are waiting needs to be considered
     carefully; does the transaction manager allow this? It is not the lock manager's decision.>

LockNoWait: PROC [trans: TransID, lock: LockID, mode: LockMode,
  procToCall: PROC [REF ANY], argForProcToCall: REF ANY]
  RETURNS [granted: BOOLEAN];
  -- Same as Lock, but never waits in lock manager. If lock can be granted without waiting, the lock is
     granted and the procedure returns TRUE. Otherwise the lock request is queued and the procedure
     returns FALSE. When the lock is probably available or lock manager times out, the lock request is
```

forgotten and the lock manger calls procToCall[argForProcToCall] with no lock manager monitors locked. (If procToCall = NIL then the reqeust is not queued and procToCall is not called. Otherwise, the idea is that procToCall just reenters whatever monitor is causing us not to use **Lock**, and notifies the condition variable where we are waiting. Upon getting this notification we can retry our call to **LockNoWait**.)

Unlock: PROC [trans: TransID, lock: LockID];
-- This means (1) error if *trans* does not hold *lock*, (2) decrement count associated with [*lock*, *trans*], if count becomes zero then release lock (and consider granting a waiting request).

UnlockTrans: PROC [trans: TransID];
-- This means for all locks held in *trans*, perform **Unlock** until count becomes zero (see description of **Unlock**).

DowngradeTrans: PROC [trans: TransID];
-- This means for all locks held in *trans*, change write or update locks to read locks. <This is a breach in the isolation of the lock manager from the meaning of lock modes; a table is required to say, for each mode, what mode it downgrades to.>

END.--**Lock**

CSL Notebook Entry

To: Alpine designers

Date: March 18, 1981

From: Mark Brown

Location: PARC/CSL

Subject: Outline of a possible FileStore implementation

File: [Ivy]<Alpine>Doc>FileStoreImplOutline0.bravo

XEROX

Attributes: informal, technical, Database, Distributed computing, Filing

Abstract: This memo contains a sketch of how FileStore might be implemented.

Priorities

Sequential I/O will always be very important, and Alpine should try hard to do a good job on sequential I/O. Applications that exhibit sequentiality: file transfer, external sorting, searching (pattern matching in text, no-index query processing), logs (both low and high level.)

There are tensions between the requirements for a server and a workstation. When in doubt, the server should take precedence in Alpine design.

FSBuffer

In the proposed design, all operations on FileStore files go through the FSBuffer interface. The interface deals in the same logical entities as Pilot's interface: files identified by unique IDs, pages within files (indexed from 0 with no "leader page"), etc. Transactions do not appear in the interface. Data is transferred across the interface in buffers that are controlled by the FSBuffer implementation. In a machine with multiple FileStores, there needs to be only one FSBuffer instance. An FSBuffer "volume" corresponds to a Pilot logical volume, but it might be implemented by duplicating all writes to two independent Pilot logical volumes.

FSBuffer can be implemented on public Pilot interfaces. It can later be improved, on basis of experience, by using Pilot internal interfaces (perhaps by doing its own buffering and disk I/O. If the implementation assumes responsibility for buffering and disk I/O, then performance can be improved by adding buffer memory to a machine, without interacting with Pilot swapper performance.)

System reliability can be improved by using replicated volumes, but this is transparent at the FSBuffer interface. (In commercial systems this replication is called "mirrored disks"; high-reliability commercial systems support replication at this level, and tolerate single failures without interruption of service.)

Basic transaction strategy

Execute file page write, file deletion, file shortening, and file immutable-making by writing a redo log entry before the end of phase one. During phase two, carry out the operation.

Execute file creation and file lengthening by writing an undo log entry, then executing the operation, before the end of phase one. Undo the operation if transaction aborts.

Optimize important special cases. For example, page writes do not need to be deferred if the file was created in this transaction.

The transaction log (See FileStoreTransCode1.bravo for more discussion of this)

The log is a stream to a file; log accesses are preformed through FSBuffer. The amount of log buffering in the FSBuffer implementation may have a large impact on the performance of update transactions.

The log file is used as a circular buffer. Pages of log are formatted so that they can be parsed even in case of an unexpected crash. One simple scheme: each page contains a sequence number that tells the number of times that the page has been written. If a page stamped "i" is followed by a page stamped "i-1", then the page stamped "i" is the last page of the log.

Log activity can be modelled as: there is one read pointer per transaction, a single write pointer shared by all transactions. The read pointers always lag the write pointer.

If protection against any single-page failure is required, then the log must be duplicated on physically separate volumes.

The log may be stored on tape for backup purposes. (Searching a lot of log tapes can be very slow; a backup tape organized for fast access is needed if recovery is to be reasonably fast.)

An interface to the log will be provided so that other programs, such as a database management system, can write their own recoverable actions.

The local map

A volatile data structure (lost in crash.)

For each transaction t, maps [file, page] -> log ID of log record containing data written by t on that page.

How various interesting operations are implemented

Write Page

Obtain page lock (U or W mode) if required.

Write log record "transaction t wrote data d on page p of file f". Not necessary to force log.

Insert entry containing log ID of this log record into t's local map.

Read Page

Obtain page lock (R mode) if required.

Consult t's local map. If page was written by t, then read data from log. Otherwise read data from file.

Commit (multiple machine, two-phase case)

Phase one.

Convert all of t's U mode locks to W mode locks.

Write log record "transaction t is ready". Force log.

Phase two.

Write log record "transaction t is committed". Force log. (Operation can return now, but processing continues.)

Enumerate t's local map. For each entry, read data from log, write data to file, and release / downgrade page lock (if any).

Release / downgrade other locks held by t.

System checkpoint

Abort any excessively long transactions.

Flush buffers of any pages that represent committed writes performed by excessively old transactions.

Write a log record that identifies (1) all active transactions, (2) the oldest log record that must be consulted to recover from the current state. Force log.

Write the restart record, making it point to the checkpoint just written. Note that restart record is duplicated to tolerate failures during write. After a crash, recovery begins by reading the restart record and finding the most recent checkpoint.

CSL Notebook Entry

DRAFT - DRAFT - DRAFT - DRAFT

To: Alpine designers

Date: March 26, 1981

From: Mark Brown

Location: PARC/CSL

Subject: Recovery in database management systems

File: [Ivy]<Alpine>Doc>DBMSRecovery.bravo

XEROX

Attributes: technical, Database, Distributed computing, Filing

Abstract: We discuss the issue of recovery in database management systems. We present several implementation schemes for recoverable actions, and explore the tradeoffs between CPU cycles, main storage, I/Os, and concurrency. One of our goals is to identify facilities that can be supplied by the underlying operating system, rather than being duplicated by each program that implements recoverable actions.

Contents

1. Transactions
Actions
Transactions and recoverable actions
Implementing recoverable actions: locking, logging, and recovery

2. Recoverable actions with physical locking
File system as simple DBMS
Buffer manager interface
DBMS using a file system's transactions and physical locking
Access patterns that perform poorly with physical locking
Fine-grained physical locking

3. Recoverable actions with logical locking
Using a shared transaction for base I/O
The recovery manager's algorithm
Writing action recovery procedures (undo, redo)
Logical and physical recoverable actions for a single transaction

4. Recoverable actions in a file system implementation
Basic recoverable file system actions
Recovering low level file structures (the VAM and VFM)
File properties
Log implementation
Graceful recovery from media failures

5. Other issues
Remote cache consistency
Transaction save points
Broken locks

6. References

1. Transactions

Actions

A database management system (DBMS) provides *actions* on the objects it implements. A DBMS typically supports objects such as *record, record set,* and *field within record,* and operations such as *create record, locate record using key, fetch value of record field,* and *assign value to record field.* (A commercial DBMS usually includes a data communications component, which supports a *terminal* object with actions such as *send message to terminal.*) These actions typically involve other low-level actions, performed on objects that the DBMS implementation hides from its client: adding a page to a file, searching a hash-addressed record set, inserting an item into a B-Tree index, etc. A typical DBMS is able to execute actions for several client programs concurrently. (Existing systems are generally *not* able to execute multiple actions for the same client program concurrently, even if multiple nodes of a network are involved.)

Transactions and recoverable actions

A *transaction* is used to make a sequence of DBMS actions *atomic*: either the entire sequence happens or it has no effect. A client program first creates a transaction by the call

```
t: Transaction ← BeginTransaction[];
```

The client then supplies t as a parameter to each action that it considers part of this transaction; the DBMS groups these actions into a single recovery unit. A *recoverable action* is any action that takes a Transaction parameter in order to join a transaction in this way.

To end the transaction t the client calls CommitTransaction[t]. If this call returns sucessfully, then all actions that were included in t have been committed. The effects of committed actions persist even in the event of a later system failure; the system provides no operation for undoing the effects of a committed transaction.

If the client gets into trouble, it calls AbortTransaction[t] which undoes all actions performed under t. The DBMS may also abort t unilaterally, for instance if t gets into a deadlock with other transactions or uses too much of some resource. (If the DBMS includes data communications, the system ensures that no message sent by an aborted transaction is received the terminal to which it was addressed.)

Besides guaranteeing the all-or-nothingness of a sequence of actions, a transaction also guarantees the *consistency* of these actions. For our purposes this means that any two transactions appear to run serially (as if there were no concurrency.)

In case of a system crash, the DBMS typically aborts an uncommitted transaction that was in progress at the time of the crash. The DBMS preserves a committed transaction in spite of the crash. (If the DBMS includes data communications, the system ensures that each message sent by a committed transaction is received at least once by the terminal to which it was addressed.)

Even the best implementation of transactions only provides the behavior just described with high probability. Carefully-designed redundacy is used to ensure that k independent failures are necessary to cause the loss of transaction integrity, for some specific k. Several commercial database management systems provide integrity in the face of any single failure of a storage device, and often survive multiple failures.

Few existing systems support the notion of transaction as we have described it. In some systems, a transaction is implicitly created by receiving a message from a terminal, and committed by sending the reply. System R supports transactions, but since no IBM file systems or other IBM data management systems support System R transactions, it is impossible to group actions not implemented by System R into a transaction. System R also does not perform message recovery or allow any parallelism within a transaction.

In our view, there is a universal notion of transaction that applies to all systems that wish to use it. We expect file systems, and general- and special-purpose database management systems to implement actions capable of taking part in a transaction. This allows any number of processes, executing on any number of machines, to perform logically related actions concurrently as part of a single transaction. The underlying operating system supports the type Transaction and the operations BeginTransaction, CommitTransaction, and AbortTransaction.

From the point of view of a *client* of a transactional file system or database management system, this is essentially all there is to know about transactions. The remaining question is: what interface should the operating system provide to the *implementor* of a recoverable action? The most effective way to ensure that a number of different systems all agree upon the same notion of transaction is to make it easy to write recoverable actions for such transactions.

Implementing recoverable actions: locking, logging, and recovery

I shall now reserve the term *database* to mean a logical storage entity which is operated on by recoverable actions. In a quiesced DBMS there is some identifiable storage volume that contains a database, in the sense that any database record can be found at some location on this volume. I shall call this storage volume the *base* of the database.

Naturally there are several ways to implement recoverable actions. Most implementations include three distinguishable but related components, responsible for *locking*, *logging*, and *recovery*.

Locking is used to ensure the consistency of concurrent transactions. LockConcepts0.bravo discusses locking issues at length. Providing an adequate interface to the lock manager does not seem very difficult. It is necessary for clients of the lock manager to lock objects that are meaningful to them, such as index intervals and individual records. The lock manager for such entities should be able to communicate with the lock manager that the file system uses to lock files and pages, in order to detect deadlocks. It is possible to manage all locks with one facility.

It is worth keeping the *lock granularity tradeoff* in mind: Locking large objects means using few locks, so the locking overhead is small, but it also means reduced concurrency. Locking small objects allows higher concurrency but is more expensive in processor cycles and storage.

Logging is used to record, in nonvolatile storage, a history of transaction executions on a database. Let the *persistent state* of a database be the state determined by all committed actions that have been executed prior to a given instant of time. Information written to the log preserves the following *write-ahead log invariant*:

> At any instant of time, the contents of the log plus the contents of the base are sufficient to define the persistent state of the database.

Since the information in a database is the result of executing a sequence of recoverable actions against an initially empty database, it follows that the complete log of a database defines the persistent state of the database at every instant of time including the present; the base is logically redundant. A system that does not compress all but the most recent portion of the log into a random-access base has unacceptable performance.

A log record typically records information at the same level of abstraction as the recoverable action that produced it. For instance "record 2583: field 3 now has value 10 because of transaction 4273" and "commit transaction 4273" are typical log records. Depending upon the timing of updates to the base, a log record produced by executing a recoverable action may contain the database values produced by the action (a *redo* log record) or the database values overwritten by the action (an *undo* log record).

As illustrated in the example above, the outcome (commit or abort) of a transaction is an event that may be recorded in a log. In a multi-database transaction coordinated through two-phase commit, there will always be some database with the property that at some instant the transaction outcome is not represented entirely within its log or base; hence the write-ahead log invariant above must be

modified somewhat. Failure of transaction outcome storage has a potentially high cost since all resources locked by a multi-database transaction are tied up until the transaction outcome is established. Lampson has proposed a multi-round voting procedure using replicated commit records to solve this problem. Replicated commit records might be implemented using the logs of several independent databases.

Logs are often implemented using sequential writes to a ring buffer of pages. But the term *log* does not imply this implementation. For instance, the Juniper intentions lists plus shadow pages constitute a log; the global map plus data pages are a base.

Recovery is a procedure invoked after a system crash. It ensures that updates performed by a transaction that was committed before the crash take effect (eventually appear in the base), and that updates performed by a transaction aborted before the crash do not take effect. Recovery may also abort transactions that were in progress (not committed or aborted) at the time of the crash.

Action recovery is generally very specific to the action; for instance, only the code that implements record storage knows how to create a record. The recovery manager is unable to interpret the base, but understands the overall syntax of the log (i.e. how to parse it into records) and can discover the outcome of transactions. The recovery manager is therefore responsible for calling action-specific routines in a sequence, determined by the transaction outcomes and the state of the log, that is designed to bring the entire system (log plus base plus volatile data structures) to a state that will allow the processing of new transactions. An action-specific recovery routine does not understand the overall syntax of the log, but can interpret a log record that was generated by the action.

2. Recoverable actions with physical locking

File system as simple DBMS

A file system may be regarded as a very simple database managment system. It supports objects such as *file* and *page within file*, and actions such as *create file, read page of file,* and *write page of file.* It is natural for these to be recoverable actions, yielding what we sometimes call a "transactional file system" (bletch.) Database management systems can build their more complicated recoverable actions on this base.

We shall not discuss the techniques for implementing recoverable file actions. Our later discussion of recoverable actions with logical locking will be relevant, however.

Buffer manager interface

A *buffer manager* interface is an effective way of providing transactional files to a DBMS implementation. It also seems to be a suitable basis for many other applications of files that involve both reading and writing.

A buffer manager combines the functions of (virtual) memory allocation and data transfer between files and virtual memory. A basic buffer manager call is "get page p of file f for readonly use by transaction t." This sets a read lock on the page (or perhaps the entire file) for the transaction, returns a page buffer containing a copy of the requested page, and increases the *share count* of the page buffer by 1. When the DBMS is through with the page, it calls the buffer manager to say "release buffer b." This merely decreases the page buffer's share count by 1; it does *not* release any locks. When the buffer's share count is 0, the buffer manager may re-use the buffer.

Since transactions do writes, the buffer manager must provide modes of access beyond readonly. The simplest way of providing modes is for the "get page" call to accept a mode parameter, which might be Read, Update, or Write (corresponding to the modes in which the page will be locked by the call.) The other way is an explicit "upgrade" call, which allows a transaction to request Update

or Write access to a page buffer for which it already has Read access. There is a subtle limitation on "upgrade" calls, however. If other transactions already have Read access to a buffer that transaction t requests in Update mode, t must wait for the reading transactions to finish. This is because the transactions are sharing memory, but the readers must be isolated from the writer. This is not a problem if a "get page" call is used, since the buffer manager has the freedom to allocate a new buffer for the exclusive use of the Update transaction. In any event, when a transaction t dirties and releases a buffer, and the buffer manager decides to re-use that buffer, the buffer manager writes the buffer to the file system under transaction t.

The buffer manager also supports miscellaneous other calls. For instance, the buffer manager will supply a buffer that is not mapped to any file. The client may wish to use the buffer as scratch space, or to write in it and then have the buffer manager map it to a specified location in a file (thereby writing the file without first reading it.) To give isolation from the implementation of files (location and replication transparency), the buffer manager should provide the complete repertoire of standard file actions such as creation, deletion, and length-changing.

The first time that the buffer manager performs an action on behalf of a transaction t, the buffer manager registers itself with the file system transaction manager. This causes the buffer manager to be notified when t is committed or aborted. When t is about to be committed, the buffer manager writes all of t's dirty pages to the file system; when t about to be aborted, the buffer manager discards all of t's dirty pages. The client may also give the buffer manager advance warning that it should begin writing the dirty pages of a particular transaction back to the file system.

A database management system implementation places two additional requirements on the buffer manager. The first requirement is that the buffer manager support a notion of *block* that may be larger than the file system's inherent page size, and be willing to buffer and transfer entire blocks instead of pages. Naturally a buffer must still consist of contiguous virtual memory, even if it is more than one page long. System R uses a block size of 4k bytes, and this is representative of commercial systems. This holds enough records that the technique of clustering of related records onto a single block (for instance, a "parent" record and all of its children) can be effective. It also gives a large fan-out from B-Tree index blocks, so that these indexes do not get too deep. The choice of block size is inevitably a compromise, and most existing buffer managers require that all buffers have equal size. If this requirement were relaxed, say to allow the buffer size to be a function of the file being buffered, the DBMS might well use different page sizes for different data structures.

The second requirement is for the buffer manager to accept pragmatic advice for read-ahead/write-behind decisions. Sequential search through a database is common. There are always queries that cannot be accelerated with an index, such as string pattern matching requests involving record fields. It is important to take advantage of sequential file access when it occurs, and it is easier for the client to inform the buffer manager than for the buffer manager to deduce the reference pattern.

DBMS using a file system's transactions and physical locking

The Cedar DBMS is built on a buffer manager similar to the one just described. All locking and recovery is done implicitly as part of file I/O requests. This has the following advantages:

Simplicity. It is very easy to write recoverable actions; all locking and recovery issues are hidden by the buffer manager. An action makes a change directly in a buffer page by LOOPHOLEing a pointer to some word in a buffer's interior into the correct Mesa type, and then storing through the pointer.

Low locking overhead. Since locks are set on relatively large physical units (files or pages), not many locks are requested by most transactions.

Distributed computation. The DBMS runs entirely on a user's workstation; no DBMS-specific code lives in the file server, so the DBMS is not a bottleneck in the system. A query that only involves pages already fetched to the workstation places no demands on the network or the file server. The file server's transaction machinery is sufficient to keep all of the remote buffer

managers consistent.

It has the following disadvantages:

Extra disk I/O. Whenever a single byte of a page is written in a transaction, the buffer manager must write the entire page to the file system.

Low concurrency. Since locks are set on relatively large physical units (files or pages), two transactions may experience a lock conflict even if they they are not operating on the same object.

Extra network I/O. The DBMS interface to the network is pages. Hence the amount of network traffic generated by a query is proportional to the amount of non-cached data looked at to evaluate the query, not the size of the query's result.

The problem of extra disk I/O only becomes serious at high rates of update transactions, so it does not seem to be a pressing issue right now. The low concurrency problem is also serious for high rates of certain classes of transactions, but may become an issue sooner for another reason. In simple interactive database applications, users experience lock conflicts as delays (seeing someone else's think time), or as transaction abortions that refresh the screen and force the user to reconsider any tentative updates. These problems can be "solved" by forcing all interactive applications to use short transactions, and to validate information at the application level before performing updates that a user has requested based on information from a previous transaction. From a logical point of view this is a great solution, but it adds complexity to applications; it amounts to application-level cache validation. By providing higher concurrency at a low level, we may make certain classes of interactive applications much easier to build. (At the same time we might explore the idea of system support for application-level cache validation.)

Access patterns that perform poorly with physical locking

Many database management systems support high-concurrency access (reading and updating) to certain important data structures, even though this means abandoning the use of a file system's physical locks (on files or pages) to control concurrent accesses.

A structure that needs to support high-concurrency access, at least in some situations, is a B-Tree index. If transactions T1 and T2 each makes an insertion that falls into the same leaf page of a B-Tree, and each independently writes this page to the file system, the file system will detect a conflict and disallow one or the other update (abort T1 or T2). Indeed, if the only point in common between the two transactions is the file server, there is little hope of avoiding this. But if the two transactions are being processed by the same instance of a DBMS, then all of the right information should be available to perform the two insertions without conflict.

Certain record accesses require high concurrency. For instance, in most database systems the catalog (the repository for data about data) is a hot spot. It is bad if a catalog entry update by one transaction (perhaps caused by a record insertion) causes a second transaction, which has read a different entry on the same page, to abort.

A third common pattern of access that is known to have bad concurrency properties is appending records to a log. A log is a simple, compact, reliable structure that is just right for a number of applications. Logs are very important in the world of commerical data processing: all financial transactions keep them. A log may be abstracted to something called a sequential-dependent record file. The relationship between records in such a file is a successor (and predecessor) function, whose ordering is determined by the time of record insertion (and not by any explicit key value in the record.) If insertion is implemented by finding the last page of the file, locking it, and creating the new record there, then the end of the file becomes a hot spot and concurrency is limited. A transaction generally requires at most that all its insertions appear adjacent in the sequential-dependent file after the transaction commits; it does not care about the identity of the record that precedes the first record it inserts. Hence a high-concurrency log implementation is possible as long as information is shared among the simultaneous transactions. (The IMS "Fast Path feature" implements high-performance sequential-dependent record files, and these play an important role in raising IMS' transaction rate from 5/sec to 100/sec for some important transactions.)

Database systems support high-concurrency access to these classes of shared structures because it increases throughput (by decreasing lock wait and I/Os.) This is true in spite of the fact that high-concurrency access greatly complicates the DBMS implementation and costs extra CPU cycles when it is used. In cases when high-concurrency access is *not* required, the DBMS will generally prefer to use the file system's physical concurrency controls (on pages or files) directly.

Fine-grained physical locking

It is possible for the file system to allow update of a byte range within a file, rather than update of an entire page or block. This is of limited usefulness to a database system. Of the three situations cited above, the only one that it may help with is high-traffic record access. It is limited to situations in which the mapping from logical update to physical update is relatively static. It also widens the buffer manager interface significantly.

My view is that a byte-range file access system is a simple data manager that should be built upon a page- or block-access file system, using the techniques described in the next section. I doubt that the number of clients for such a system is large, however.

3. Recoverable actions with logical locking

We now consider the construction of a DBMS with logical locking. We shall build on a transactional file system, as in the previous section, but will use file system transactions in a much different way.

Using a shared transaction for base I/O

Our basic strategy is to use a single transaction s for all page I/Os to the base of a given shared database. A transaction t that performs an update to this shared database will lock the updated portion for transaction t and write a log record describing the update and identifying transaction t. At some point (before or after t commits) we make the base update in the buffer pool under transaction s.

A CheckpointTransaction[s] call, which has the effect of committing updates made by transaction s and downgrading all of s's write locks to read locks, is useful (in systems that don't support this as a primitive we must commit s and start a new transaction, discarding the state of the buffer pool.) We checkpoint transaction s occasionally in order to commit changes to the base and thereby bound the amount of work required at restart. These checkpoints are taken only when no update actions are in progress. This restriction means that the committed base is always action-consistent: it is the correct result of performing some sequence of actions, without reference to transactions. It is this restriction, which greatly reduces the possible states of the base at restart, that gives us a benefit from the underlying transaction mechanism. Note that to satisfy the write-ahead log invariant, the log records for all updates made to the shared structure must appear on the disk before s is checkpointed. This invariant is easy to satisfy if the file system and DBMS logs are implemented as a single log, since s's checkpoint will force the combined log to disk.

There are two specific versions of this strategy, which we call *in-place update* and *deferred update.* The in-place update strategy is to acquire a write lock, write the log record containing both undo and redo information, and then perform the update in the buffer pool, *before t is committed.* Because of the continuous flow of transactions, it is not practical to exclude the possibility of checkpointing s before t commits. Thus there is never any guarantee that base, or even a single page of the base, is consistent with respect to any particular transaction that is using it; only locks prevent the multiple transactions from seeing each other's updates.

It is possible to avoid logging redo information in the in-place update strategy by checkpointing s immediately before committing any transaction that made updates under s. As a practical matter

this is probably a poor strategy, because the amount of extra log I/O and computation for the redo record is negligible compared to the I/O cost of checkpointing s.

The deferred update strategy is to obtain an update lock, write the log record containing redo information, and then perform the update in a volatile data structure (sometimes called the *shadow.*) During phase one of commit, we convert the update lock to a write lock, and some time after the start of phase two we perform the update in the buffer pool. In this case it is possible to serialize the updates of committing transactions, so inconsistent states of the base are not necessarily commited. But there may be no advantage in serializing updates; it depends on the application.

In comparing the two variants, note that in-place update requires both undo and redo recovery routines, and uses a write lock from the time of update until commit. Deferred update requires only redo recovery, which may be significantly simpler than undo and redo, and holds a write lock only during phase one. The locking advantage is probably not significant. But the disadvantage of deferred update is special code to maintain the shadow data structure, and more costly lookups since the shadow structure must be consulted.

There seem to be applications for both techniques. Of the three examples listed above, B-Tree and record update seem well suited to in-place processing. But the sequential-dependent record file is probably better implemented using deferred update, since undo is very clumsy and the shadow data structure can be simply a list of log record IDs.

It appears that no tight coupling is required between a file system that provides transactions involving file pages and a DBMS that provides transactions involving database objects. It is interesting to note that System R, which uses the in-place update strategy described above, became tightly coupled to its file system. The System R designers now say that their file system is inappropriate for large databases (> 100 mByte), but they cannot change it without a major rewrite to System R.

The recovery manager's algorithm

After a crash, the recovery manager first recovers the file system to a state that is transaction-consistent with respect to file actions. In the process it determines the outcome (committed, aborted, or ready = finished phase one) of each transaction that has been active since the last file system checkpoint.

The recovery manager then restarts all *resource managers* (subsystems that write log records and perform undo and redo), and begins to interpret the log of DBMS actions. <During its restart a resource manager might acquire an exclusive lock on a file that it "owns">. First it undoes aborted transactions, in reverse chronological order. This means: for each undo log record, call the action recovery undo procedure, passing it a copy of the log record. Then it redoes the ready and committed transactions, in chronological order. This means: for each redo log record, call the action recovery redo procedure, passing it a copy of the log record. At the point in the log at which "ready" or "committed" is determined, the recovery manager informs each resource manager of this fact. <Is locking off during recovery, as in System R? This is an decision that each resource manager is free to make. Most of them will probably take this approach. I don't yet understand the issue of locking and deadlock during undo.>

Recovery routines have all of the facilities of a transactional file system at their disposal. They may create, commit and abort file system transactions. <What about writing more log records? Probably a bad idea.>

Since recovery may fail, the same action (log record) may be redone or undone several times. But it never happens that an action is first redone and later undone, or vice-versa.

Writing action recovery procedures (undo, redo)

The most difficult part of implementing fine-grained update seems to be in designing the data structure so that recovery can be performed from any action-consistent state. An action recovery procedrue (undo, redo) must be *idempotent*: executing it (fully or partially) any number of times must have the same effect as executing it once.

A basic implementation technique is to version-number objects that are subject to updates. Log record IDs make good version numbers in this application. Let a data object d contain the ID of the log record for the most recent update applied to d; call this d's *update ID*. Since applying an update modifies the update ID, an undo log record contains the previous value of the update ID.

During recovery, a redo recovery routine will first read the update ID of the referenced data object. If the update ID is less than or equal to the log record ID of the redo log entry, then the redo is applied, otherwise it is skipped. Similar logic applies to undo.

If all updates to an object are undo-logged, and log record IDs are used as update IDs, then the combination of object plus log has a very simple structure: the object points to the most recent update, which points to the previous (committed) update, etc. The entire history of the object can be found by traversing the log in this way. Because of locking, only the most recent update is possibly uncommitted and hence candidate for undoing. Undo pops the top element from the stack of log records.

Logical and physical recoverable actions for a single transaction

The point of view that emerges from the preceding discussion is that a DBMS action for a transaction t should, whenever possible, be performed by simply using file actions under transaction t. In cases where this leads to unacceptable performance, a shared transaction should be used for the file actions and logical locking and recovery procedures should be associated with the DBMS action. Note however that if t commits and the system crashes before the shared transaction commits, the file system will recover t's updates that were done as file actions, but will not recover t's updates that were done under the shared transaction. So the limitations of "action-consistent" must be understood by those who write logical update recovery procedures. Only updates covered by the shared transaction are guaranteed to be consistent. Hence recovery will be *much simpler* if it is based entirely on information that is only updated under the shared transaction. Data structures that are subject to these shared updates must be designed with this local recovery property in mind.

4. Recoverable actions in a file system implementation

We noted earlier that a transactional file system may be regarded as a simple DBMS. We now wish to discover what implementation techniques carry over directly from the general DBMS case to the file system case.

Our DBMS' are built on file system transactions. Our file system is built upon page I/Os, perhaps duplexed (pages written twice on independent media) for redundancy. A duplexed page write behaves like a simple transaction, since a failure during the first write can be undone by reading the second page, and a failure during the second write can be redone by reading from the first page. A non-duplexed page write does not behave like a transaction, because there is no (local) way to recover from a failure.

In any case, file page writes are very limited transactions. Hence there is no hope of structuring the file system implementation along the lines of our DBMS that used the transactional file system for its recovery. Instead, the file system implementation must be structured along the lines of the DBMS with a shared transaction.

Basic recoverable file system actions

Four file system actions seem to be typical of all problems involved in the implementation. They are *create file*, *destroy file*, *read page of file*, and *write page of file*. The action *set file size* is analogous to *create file* if the file is being extended, and analogous to *destroy file* if the file is being contracted.

It is natural to implement *create file* using an in-place update strategy. The reason is that *create file* must acquire resources, and to commit a *create file* request we must be sure that the resources are truly available. A straightforward method to ensure this is to actually carry out the action, then undo it if the transaction aborts. For similar reasons it is natural to use a deferred update strategy for *destroy file*. If a destroy file action is aborted, then the "destroyed" file must be preserved. A straightforward method to ensure this is to defer the action (keeping a volatile record of having done it), then do it during phase two if the transaction commits.

The *write page of file* action has several possible implementations, each with different properties. Since reads dominate writes in nearly all applications, the write action cannot be evaluated without considering the *read page of file* action.

An in-place update *write page of file* implementation requires undo logging: the entire old contents of a base page must be logged before the base page is updated. This costs a base page read if the old contents of the base page are not in the file system buffer pool. In the simplest implementation a writer must wait until (1) the old base page contents are read, if necessary, (2) the old base page contents are written into the log, and (3) the new base page contents are written. So the real-time delay for a write is considerable in the obvious implementation. This can be improved by acknowledging the write immediately and actually processing it in background; this may be considerably more complex to implement (consider the synchronization at commit time and when a transaction reads a page that it has written.) Note that here, unlike the DBMS case, it may make sense to "checkpoint the shared transaction" (force all base page writes) before committing a transaction, since this avoids the cost of redo logging. If we follow this strategy then a write that is acknowledged before it completes may cause a real-time delay at commit time. Ignoring writes that are acnowledged before they are completed, the *read page of file* action is straightforward.

There are two possible deferred update implementations. The first is to use shadow pages for writes, log only pointers to the shadow pages, and swing the pointers after commit. This approach means that sequential in the file is not sequential on the disk; hence sequential access performance is poor. The complexity of this approach is also relatively high due to the management of shadow space (how do you avoid scavenging the disk to reclaim shadow after a crash?)

The alternative deferred update approach is to log the entire new contents of a page before commit, and to perform all updates after commit. This requires a lookup in a volatile data structure for each page read, to see if the data is in the log instead of in the base, but generally the data will be in the base or be buffered in the log. It has the attractive properties that (1) once the log write has been buffered and the volatile map entry made, a write action may return, (2) commit requires only a log force, and no synchronization with base writes, (3) sequentiality of file pages is preserved. Like the pure undo scheme, three I/Os are required in the worst case: a log write, a log read, and a base write. For short transactions the log read will be buffered in the file system buffer pool and hence will be cheap.

Important special cases can be optimized in any of these approaches. For a file server an important special case is writing a file in the same transaction that created it. In this case there is no need to defer updates since the file will disappear if the transaction is aborted.

Recovering low level file structures (the VAM and VFM)

A file system contains certain permanent data structures that are not visible to the client. One is the volume allocation map (VAM), a mapping from disk page address to {free, occupied}. Another is the volume file map, a map from [file unique name, page in file] to disk page address. If updates to

these maps are not recoverable then a potentially costly scavenging operation may be necessary when a file system crashes.

The file system can ensure that updates to these maps are recoverable by following a write-ahead log protocol on such updates.

File properties

File properties such as create date, etc should be built on top of file system transactions, the way that DBMS objects are.

Log implementation

The log must be simple and robust. It is typically imlemented as a single large file, allocated in consecutive disk pages (modulo bad spots.) If protection against single-page failures is needed, then the log must be duplexed. By saving log pages on tape before they are reused, we ensure that even old data can be recovered.

There are several arguments in favor of merging the logically separate file system and DBMS transaction logs into a single log. These include (1) simpler synchronization of log forces, (2) reduced disk arm motion, (3) only one resource to run out of, (4) no merging required when saving log to tape. There are undoubtedly situations in which each of these turns from an advantage to a disadvantage, but on balance we like a single log.

Graceful recovery from crashes and media failures

Note the tradeoff between the detail of logging and the (potential) cost of undo and redo. Suppose that a write page action changes a single byte of a page. For redo, it is logically sufficient to simply log the page number, byte offset, and new byte value. But in the event of a crash, the event of a crash, the page may not be readable. So in this case redo must reconstruct the previous state of the page somehow (perhaps by playing the log forward from a backup copy of the file system), and then apply the update.

A better strategy, in terms of recovery time, is to redo log the entire contents of the page, with the update applied. Then redo simply rewrites the page. The number of bytes written to the log is larger for this strategy than for the previous example. But *unless either the entire old contents or entire new contents of a page are logged* before a page is updated, a failure in the middle of updating the page can force a recovery whose cost may be quite high. This is the motivation for full-page logging for all page updates.

<Media failure: duplexed storage case; recovery from backup and log case. Unclear how to best organize system so that media failure does not make entire system unavailable (ok to delay the transaction that recovery is in behalf of.)>

5. Other issues

Remote cache consistency

<Caching reduces load on network and file server and improves response. But updates are not broadcast to all caches. So server (who does see updates) is responsible for maintaining cache consistency.>

<A plausible page-level protocol based on file version numbers and recent update history.>

<Cache consistency between two agents of the same transaction -- a client problem.>

Transaction save points

<How they are used (and not used) in System R. How to implement them in a log-based system.>

Broken locks

<How they can be used to hide transaction aborts from higher-level clients.>

<Multi-client transactions are a necessity for a scheme with broken read locks and lock releasing. Explain why.>

6. References

Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger, "The recovery manager of a Data Management System," IBM San Jose Research Report RJ 2623. (This very nice paper describes the System R recovery manager. It does not give a very coherent explanation of the relationship between System R transactions and the transactions provided by its underlying file system, or explore alternatives to the many design choices made in System R.)

Raymond A. Lorie, "Physical Integrity in a Large Segmented Database," *ACM Transactions on Database Systems 2, 1* (March 1977), p. 91-104. (Describes the interface and implementation of the file system used by System R.)

Mark Brown, [Ivy]<Alpine>Doc>LockConcepts0.bravo. (Describes the scope of the lock manager of for file and database systems.)

Butler Lampson, [Ivy]<CSL-Notebook>??>81CSLN-0001.press. (Describes replicated commit.)