A Proposal for a Modula-2+ Programming Environment

John R. Ellis, Last modified on Mon Jun 1 00:03:14 1987 by ellis

1. Introduction

This proposal attempts to sketch out a framework for a new Modula-2+ systems programming environment. This environment would replace the current one in about 18 months and would serve SRC for about 5 years.

How will this new environment be different from what we have now? The various meetings held in the last six months have repeatedly compiled wish lists, and from these lists I've selected those goals which I think are both desirable and feasible:

System modeling (already under way).

Very fast turn-around in the entire development cycle, even for interface changes.

Structured editing, browsing, querying, and debugging via an integrated user interface.

Programmatic manipulation and generation of Modula-2+ programs, including a uniform representation of types.

Dynamic loading fully integrated with debugging and system modeling.

Better runtime performance: fast procedure calls, faster program startup, minor improvements in generated code, shared libraries, and allowance for classical optimizations.

A code generator for both the VAX and load-store machines like the PRISM.

Is this research or advanced development? Some of both. No industrial-strength environment yet provides system modeling, fast turnaround, structured editing, and programmatic manipulation of programs. But to demonstrate that our environment really is industrial strength (or could be made so), we'll have to invest development effort in basic components like code generators and shared libraries. Each individual component may seem quite pedestrian; the interesting research is

in assembling these components into a coherent, workable whole.

It is not a goal of this proposal to:

Support more than one systems programming language, including Pebble (though prototype language implementations may well be built on top of the environment).

Support the Unix/C style of small programs invoked very quickly from the shell (though parts of this proposal will improve start-up time).

Generate super-efficient code.

I've made no general attempt to describe the space of possible design choices or why I made many choices; done correctly, that would take many more months and fill many volumes. I hope that this proposal will provide a concrete starting point for future discussions.

2. Fast turnaround

In the committee meetings last fall, fast turnaround in the edit-compile-link-execute-debug cycle was the number one or two item mentioned by most of the attendees. A 10-second turnaround for one-line changes would greatly improve our productivity, make us all a lot less grumpy, and make the construction of small and large programs more pleasant.

In our current environment, the slow turnaround is dominated by one factor: the large number of bytes being fondled at each stage of processing. SRC is not peculiar in having large libraries and binaries. It is inevitable that high aspirations (texts, threads, user-interface toolkits, RPC, source-level debugging, etc.) will lead to large binaries. From conversations with people at CMU, Sun, and other institutions using C, I've gathered that they are suffering the same problems of huge libraries and binaries and slow turnaround.

Byte fondling can be reduced using two basic strategies: retaining expensive-to-compute or frequently used information and regenerating less frequently used information on demand. More specifically, the following will reduce fondling:

An editor, compiler, binder, system modeler, and debugger in one retained address space per development session, sharing a cache of source and object.

Procedure-at-a-time recompilation.

A new method of binding (linking) object modules.

Shared libraries.

Regeneration of most "symbol table" information on demand during a debugging session.

Compiled definition modules.

Fine-grained inter-module dependency analysis to reduce recompilations.

Dynamic loading of modules ("load and go"). A program image

normally won't be written into a file during the development cycle.

3. Runtime architecture

The architecture of running Modula-2+ programs has a direct bearing on fast turnaround, shared libraries, dynamic loading, and the simplicity of implementation.

This proposal entails a completely new runtime architecture for M2+ programs with only the barest minimum of compatibility with the current architecture implicitly defined by the current compiler, linker, and Ultrix conventions.

There are two key ideas in the new architecture:

> Representing the runtime structure of a program as safe REF structure with a simple M2+ interface.
>
> Using true interface records to implement binding.

A safe REF structure has several advantages. A highly optimized utility (Pickles) is responsible for storing representations on disk. The output of the compiler is almost exactly the runtime representation of a module. The editor/compiler can easily make incremental changes to a structure without worrying about storage management. Large applications can include their own initialized REF structures (much as we use Bag today), without incurring any startup cost. For example, an application could bind in a large "vbtkit" dialogue or the initial Tinylisp symbol table of 800 names.

Using true interface records greatly simplifies the whole issue of binding. An interface record is simply a record containing the procedures and variables exported by an interface. To bind a client to an interface implementation, one simply stores a pointer to the interface record in a known location in the client. The client references the interface's procedures and variables by dereferencing the record pointer.

The binding that occurs during binding, shared library startup, and dynamic loading is provided by a single, very cheap operation. The not-insignificant complexity of traditional binders, especially with regards to efficiency and incremental dynamic binding, disappears almost entirely. Binding and dynamic loading can occur virtually at disk speed. During program development, it won't be necessary to actually create a binary image of the program each time through the cycle; instead, the individual modules could be dynamically loaded into a fresh address space one at a time.

4. Can we afford delayed binding?

Of course, interface records impose an extra memory reference on every inter-module procedure and variable reference. Can we afford that?

In our current M2+ implementation, 25-30% of execution time is spent in the calling sequence, and the average call takes about 20 usecs. An extra memory reference for the indirection will cost 1 usec, so we've increased the cost of a call by 5%, increasing total execution time by at most 1.25%. Not all

procedure calls are inter-module calls, so the actual increase will be less. I don't have good estimates of the usage patterns of global variables, but my guess is that the effect on them would be even less than for procedures. (A very simple, specialized common-subexpression analysis could identify multiple references to an interface record pointer.)

So we could afford delayed binding in the current environment. But don't fall into the trap of saying, "Ah, if you're speeding up procedure calls by a factor of two, three, or more in the new environment, then the relative cost of delayed binding will be higher". That's true, but mostly irrelevant. We're mainly interested in a holistic comparison of the current environment with the new one.

Overall, programs will run faster in the new environment due to much faster procedure calls, slightly better non-call code, and reduced demand on VM through shared libraries. We'll be sacrificing a small part of the potential improvement for the benefit of faster turn around and a much simpler implementation. As was pointed out in the SRC review, one of the ways we can profitably use the increasing number of cycles is to delay binding.

5. What's in a program?

The runtime representation of a program is a set of modules plus some auxiliary information. Each module contains:

It's name (a modeler UID of some sort).

The interface records exported by the module.

A linkage vector containing the non-immediate values referenced by the procedures, including imported interface records, initial REF values, and space for global variables (exported variables are in the exported interface records).

The procedure code.

The runtime representation of types (REF structures) needed by the GC, pickler, and other tools.

Any other initial REFs provided by the program itself.

In addition, each program has a global module table to support dynamic loading.

All other information needed for debugging and incremental recompilation is kept in the abstract syntax tree produced by the compiler; the AST is not part of the runtime representation. (The AST is described in a later section.)

6. Binding program images

To build a program image (a single file of bound modules that can be loaded and started quickly), the binder dynamically loads (unpickles) each module, creating an in-memory structure that is exactly the runtime representation of a program. It then pickles the entire image into a file, relocating all pointers relative to a given starting address. That file can then be read into memory at the given starting address and executed

without any additional relocation. (Adding the "pickle to absolute address" feature to Pickles should be straightforward.)

For minimal compatibility with the existing Topaz and Unix, program images are stored in a.out format. But all code and data is stored in the data segment, and the other segments are empty.

As the top-level procedures and variables in the image are pickled, their location is recorded in a PC map written at the end of the image. This PC map is used by the debugger and the exception mechanism.

All code is entirely position independent, needing no internal relocation. Some sort of base-register scheme implemented in the calling sequence (needed for PRISM anyway) allows the code to access the module's linkage vector.

There is no "library search" in the traditional sense. Given a system model, the system modeler and the M2+ modeler bridge conspire to provide the binder with an exact list of modules to be bound into an image. (Of course, the modeler and bridge will have to compute that list very efficiently to support fast turnaround.)

7. Shared libraries

A shared library is essentially a partial program image, that is, a set of modules bound and relocated to an absolute address. When a program starts up, it maps a shared library into its address space copy-on-write and dynamically binds to the interfaces exported by the library, using the same simple binding operation used for image binding and dynamic loading. Individual modules in the library are then initialized by the standard init-procedure mechanism.

Support for multiple shared libraries is essential. RPC, srclib, the vbt toolkit, and Tinylisp would each form a separate library. About 4% of the address space (170 megabytes) will be reserved for the libraries. Each library will be allocated a fixed portion of the library space based on its current size plus a generous allowance for growth. The allocation will be maintained manually by some Library Czar. He ensures that any two libraries that could be mapped into the same program don't overlap.

When a program requests that a library be mapped in, which version will it get? Given that we will have several libraries, each one undergoing continual development, it's important that we automate version control without inhibiting the growth of interfaces.

A program is compiled and bound against particular versions of a library's interfaces that were specified in the system model describing the library. The modeler and the Modula-2+ modeler bridge conspire to place a stub in the program that, at startup, will map in an implementation of the library that is type-consistent (in the modeler sense) with the interfaces the program was compiled with.

In the dumbest implementation, we would just map in an implementation of the library that provided exactly the interface versions imported by the program. But this isn't acceptable, since libraries are likely to be changing frequently

(say once a week); thus we would lose much of the performance benefit by having too many versions of the library mapped in at once.

But mapping in a library is just a type of binding. For all binding operations, no matter when they occur (building an image, mapping a library, dynamic loading), we use fine-grained dependency keys to decide when it's safe to bind implementation modules compiled with one version of an interface with client modules compiled with another (see the later section on fine-grained dependency).

When a program starts up, its stub presents the operating system with a set of pairs <name, key> representing those names which it actually uses from a library's interfaces. A "key" is the fingerprint or unique id representing the name's definition. The operating system will look for a version of the library that is already loaded whose pair set includes the set requested by the program and that is no older than the program itself. If no such version is found, the exact version in existence at bind time will be fetched from the file system.

An envelope calculation shows that the set representing "srclib"'s interfaces could be represented in less than 20K bytes. Of course, a typical program imports only some of the names exported by "srclib", so the pair set it presents will be much smaller. The subset check needn't examine all pairs in the set; the key for an entire interface is checked first, and only if that differs will the individual name pairs be checked.

8. Interface records and incremental recompilation

Compiled clients refer to particular names exported by an interface by indexing into its interface record. The simplest method of assigning offsets to names would be to number the names sequentially. Unfortunately, clients would have to be recompiled whenever the numbering changes, such as when a new procedure is inserted in the middle of an interface.

To get around the renumbering problem, the incremental editor/compiler will explicitly manage the set of assigned offsets for an interface. Once a name set is assigned an offset, it retains that offset through successive changes and recompilations of the interface. If a name is deleted, its offset can be reused. A name's offset will be part of its compile-time value used in the dependency analysis, so if an interface happens to be recompiled from scratch and its names renumbered, clients will be forced to be recompiled.

9. Compatibility with Ultrix

This proposal provides only a bare minimum of compatibility with the "standard" Ultrix runtime architecture implicitly defined by the language processors, the assembler, the loader, and the kernel. Only one or two of SRC's projects (e.g import X windows) may need to bind large amounts of foreign C into a M2+ program.

Of course, Topaz will continue to run standard Ultrix binaries.

Modula-2+ images will run on Ultrix, but shared libraries won't

be implemented. There will be almost no debugging facilities on Ultrix; stack trace-back plus the fast-turnaround insertion of print statements will suffice for those few who need to do Ultrix debugging.

C and assembler code purporting to export a M2+ interface can be bound into program images through the use of a .o converter. The C or assembler code must use the M2+ conventions about interface records. For example, C might access an imported interface record by doing:

```
(*Wr->PutText)(...)
```

where "Wr" is a global variable that the .o converter is told represents an imported interface record. Except for imported M2+ interfaces, the foreign object module must be completely self-contained (one can always use "ld" to combine several .o's).

The debugger will not know much, if anything, about foreign modules. Such modules must be well-behaved with regards to the M2+ runtime architecture; for example, they can't expect to tromp through memory. We'll probably need to supply our own versions of the necessary C library routines to support things like memory allocation.

The core M2+ system itself will contain no Ultrix .o's, being written entirely in M2+ and assembler.

10. Public interface for manipulating program source

An important goal of this proposal is to support the structured manipulation of program source at the module level and below. (The system modeler provides operations above the module level.) We have many applications that do such manipulation already: The editor, the compiler, the debugger, the pretty printer, flume, the Tinylisp stub generator. Future applications include: the system remodeler, the definition-use database, source-based profiling and testing tools, Larch, perhaps a Web-like tool for integrated documentation and source. If we could provide a single interface that supported all of these applications, we would have smaller, more robust tools with greater functionality. In addition, such an interface would make it easier to experiment with new tools.

The danger in using a single interface is inefficiency: Attempting to provide generality or simplicity often conflicts with the desire for utmost efficiency in critical components like the compiler. In this instance, I think it is possible to reach a workable balance. We'll introduce some inefficiency into the interface but more than make up for it by doing fine-grain dependency analysis and procedure-at-a-time recompilation, greatly reducing the total development-cycle time compared to the present.

The interface will be designed for the immediate potential clients (the compiler, the debugger, the pretty printer, and flume). It will not necessarily be compatible with current implementations of those clients. Only after the first design is complete will work on the compiler begin (and of course, there will be refinements or complete redesigns during the course of implementation.)

The interface will provide a decorated abstract syntax tree

(AST) representing a module. An abstract syntax tree is simpler than strict parse tree for a particular grammar, but it is close enough to the concrete syntax to allow the source text to be faithfully regenerated. Most clients won't need all the information used by the compiler, so the AST will be accessible after each of these main phases:

> After parsing, with retained comments options. (Pretty printer, editor)
>
> After name resolution and type checking. Name resolution binds each occurrence of an identifier to its definition, and type checking assigns a type to each name and expression. (System remodeler, source-based profilers)
>
> After complete code generation.

Choosing an actual representation will be a tricky balancing act between efficiency, generality, and simplicity. AST nodes will be defined as M2+ REF records with some fields marked as private to the AST implementation. A standard object-oriented vector-of-procedures will be provided for clients who desire simplicity over efficiency. For example, Type(node) would return the type of any node. Some kind of method-list scheme will provide extensibility for individual clients. The compiler itself will probably short-circuit the object-oriented interface and reference the node records directly. (Unlike many object-oriented applications, the number of distinct node types supported by the interface is fixed.)

There will be no "symbol table" in the traditional sense. Each AST node defining a new lexical scope (module, procedure body, record definition) will have a map from names declared in the scope to the AST nodes representing their definitions. All information about a name is stored in the AST node for the definition. The name and type resolution phase binds each use of a name (leaf identifier nodes) to the corresponding definition node.

Names from imported interfaces will be handled the same way. An imported name will be bound to the AST node representing its definition in the interface. This AST node is obtained from the previously compiled interface.

The AST representation for a type definition will also be used as the runtime representation of types used by the GC, pickles, the debugger, etc. (Of course, the GC will still have its specialized structures inside the type representation.) This approach would facilitate a "runtime type" mechanism like Cedar's that allows more general introspection by programs; but this is not an immediate goal of this proposal.

11. Object files, recompilation, and debugging

Compiling a module results in two structures, an AST and a "runtime unit" that is exactly the runtime representation of a module. Both are REF structures that will be pickled into separate output files stored by the system modeler. Compiled definition modules have an AST only.

Past experience with programming environment projects shows that fully decorated ASTs are very large and that writing and reading them is very expensive. But for our purposes, it isn't necessary to store the entire AST. Instead, the sub-trees

representing procedures bodies will be niled out when the AST is pickled, greatly reducing its size.

When a client such as the editor or debugger needs the AST for a procedure body, the AST interface will regenerate it on the fly (if necessary) by recompiling the procedure. (Each AST node contains an offset into the immutable source file.)

The immutable source files and unpickled object and AST files will be cached in cooperation with the modeler. The editor, compiler, and debugger will be packages bound into a single address space sharing the same cache, and that address space will be retained during an entire development session. Thus, fully recompiling an average procedure should take less than a second (Lightspeed Pascal compiles at 200 lines/second).

Normally, the detailed debugging information for individual procedures is neither generated during compilation nor stored in the AST files. Instead, if the debugger needs the PC to AST node map or the PC to variable location map for a particular procedure, it will regenerate code for the procedure, asking the compiler to retain debugger information. The modeler guarantees that the correct source will be used.

Currently, only a tiny fraction of such debugging information is ever used during the lifetime of a program. By regenerating it on demand, we'll greatly reduce byte fondling, speeding up the overall development cycle and debugging in particular.

12. Fine-grained dependency

A single fine-grain dependency analysis will be used to reduce compilations due to interface changes and to changes within a single module.

A straightforward approach to dependency analysis would be to record in each compiled definition the attributes of external names used in the compilation of that definition. For example, if a procedure P referenced some type T, the the full definition of T (including any definitions T depends on) would be recorded in the compiled P. To see if P needed recompiling, one would compare the recorded definition of T with its current definition.

Greg Nelson has generalized such a scheme in his Juno proposal. But there's no need for such generality here, because there is essentially only one attribute that figures in dependency analysis: the compile-time value of an external definition. The compile-time value of a type is its definition, the compile-time value of a constant is its value, the compile-time value of a procedure or variable is its type.

Recording the entire value of an external name in each dependent definition is probably too costly. Instead, some kind of fingerprint or unique id representing the value could be used. The fingerprint would be computed from the AST, not the source text, filtering out changes to whitespace or comments. Alternatively, one could assign a new unique id to a definition each time its AST changes.

The system modeler knows only about coarse-grain dependencies of the form "module M depends on interface I". When I changes, the modeler will ask the M2+ compiler to recompile M; the compiler will check the top-level definitions of M and recompile only

those whose dependencies have in fact changed. Similarly, when the user edits some of the top-level definitions inside M, the compiler will check all the top-level definitions for those whose dependencies have changed.

13. Structured editing

The user interface (the "editor") will provide integrated access to:

the system modeler;

editing of program source;

incremental compilation and binding;

use-definition and definition-use queries (show me the source for procedure X; show me all uses of procedure X);

the display of source by the debugger and other interactive tools.

The Smalltalk model of macro-structure editing will be used: explicit structure commands manipulate large-grained components, while traditional text editing is used to modify small-grained components (procedures and other top-level definitions in a module). Tree-based commands will *not* be used to edit procedure bodies.

The large-grained structure hierarchy is:

```
system model
    module
        top-level definition in a module
```

On top of this structure are the use-definition and definition-use information.

Viewing and editing programs are always relative to some root system model (for example, "Topaz version 10,193.0"). The system modeler "control panel" will be used to select, construct, and manipulate system models. While models can be represented as a traditional textual lambda calculus, most users will never need to read or edit that representation; instead they'll use the control panel's dialogue-style interface. The few wizards who need to edit the lambda calculus directly will use a vanilla text editor.

What will the user interface for editing modules look like? Two models come to mind, which I'll call the "pane model" and the "elision model". In the pane model (used by the Smalltalk browser), vertical and horizontal collections of panes or menus display the names of components at each level of the hierarchy, with a larger pane for displaying and editing a selected leaf. In the elision model, nodes of the hierarchy are displayed vertically in pre-order; nodes are either elided completely, abbreviated into a line or two, or else expanded into their full definition.

Discussions of which model is most appropriate are still mostly religious, based on limited experience in highly constrained environments. I don't have a well-formed opinion on the issue, and whoever signs up for this will have to deal with SRC's religions in addition to designing an interface. However, the

basic functionality provided by the user interface is mostly orthogonal to the religion.

The implementation of the editor will manipulate the structure of a module using the abstract syntax tree provided by the AST interface (which includes the compiler). The source for a module will be stored as a simple text file, while its AST will be pickled. Both files are immutable, managed by the system modeler; from the modeler's point of view, the AST is a derived object.

When the editor reads a module's source file, usually its AST will already exist, and the editor will read it as well. If the AST doesn't exist, the editor creates one by parsing (but not typechecking or generating code).

The AST interface provides a mapping from AST node to source text and vice versa. This lets the user interact with the tools via the source text, while the tools themselves deal only with the AST.

When the user edits the text of a top-level definition, the editor automatically invalidates the corresponding sub-tree in the AST, recompiling it on demand (such as when the user says "go"). Of course, compilation errors will be displayed by highlighting the source, Since compiling a procedure will be very fast (less than a second), the compiler's error handling can be much simpler -- it just reports the first error and then give up. The user fixes the error and says "go" again.

To pretty print a procedure, the editor simply invokes the pretty printer package, passing it the procedure's AST and getting back the formatted text.

The debugger will use the editor to display source and ask the user for breakpoint locations. The debugger will use the AST interface to compute variable locations and source positions from PC values and to compute PC values from source positions. The debugger lives in the same address space as the editor, sharing the same cache of ASTs.

To support browsing and querying, the editor accesses the use-definition information directly from the AST. After name resolution and typechecking (relative to some system model), each identifier node has a direct reference to its definition node, whether that identifier is defined locally or imported. To implement "show definition", the editor maps the cursor position onto an identifier AST node and then displays its definition node.

The inversion of the use-definition relation, definition-use, is provided by a separate interface to a tool that computes that information in background, similar to the current prototype application for Rdb. The implementation will use the AST interface to extract the use-definition-model triples and store them in its database.

Ivy will be used as the text editor, and perhaps Ivy's simple forms can be used for the macro-structured editing interface. It is not a goal of this proposal to support multiple editors (getting a single good one will be hard enough).

14. Compiler
~~~~~~~~~~~~
~~~~~~~~~~~~

The "compiler" is merely a package that implements the AST interface. The compiler is bound into the same address space as the editor, the binder, the debugger, and perhaps the system modeler, sharing the cache of source and object. There will be nothing new or fancy about the actual compilation algorithms; what is different is the AST interface, the ability to do procedure-at-a-time recompilation, and the use of pickles to store the ASTs.

Classical procedure-at-a-time optimizations such as common-subexpression elimination and loop-induction-variable simplification will probably not be implemented, though the compiler design will allow for them. Optimizers are still expensive to build and maintain, and their overall marginal benefit in our systems environment is small; the extra 10-15% speedup probably isn't worth the man-year of effort. And no one has expressed interest in mounting a full-scale research effort in "super compilers" (which typically involves several full-time people).

However, careful attention will be paid to basic code generation, procedure calls, and register allocation (essential for fast procedure calls).

The same code generator will be used for both the VAX and the PRISM. It will generate pseudo-instructions for a load-store machine model, and a machine-specific peephole optimizer will convert the pseudo-instructions into actual VAX or PRISM code. Treating the VAX as a load-store machine instead of attempting to use its complicated addressing modes will not only simplify code generation but probably result in faster code. (See Dick Sites's memo to VAX compiler writers.)

The register allocation will be done on-the-fly during code generation, using a simple priority scheme to choose values that need to be spilled to memory. Actual arguments to procedure calls will be targeted into the desired argument register. Unlike the current compiler, all values will by default be assigned to registers, not stack frames. On-the-fly allocation works well because the number of live values referenced between procedure calls in systems programs is small, and the caller-saves calling sequence forces all live registers into memory.

No attempt will be made to do inter-procedural register allocation. David Wall's experiments showed that, for his benchmarks of large systems programs on a 56-register machine, interprocedural register allocation (with arguments passed in the registers) was only 10 to 20% better than procedure-at-a-time allocation (with arguments *passed on the stack*). It's likely that much of that improvement was due simply to passing arguments in registers, and that a fast calling sequence and programmer-specified procedure integration will gain most of the benefit of inter-procedural register allocation.

15. Calling sequence

Our programs currently spend 25% of their time in the calling sequence. By switching to a minimalist calling sequence using already tested technology, we could get a two- to four-times across-the-board speedup in procedure calls. The same calling sequence will be used for both the VAX and load-store machines like the PRISM. See my message on src.mp of 3/19/86 for a

detailed discussion. (I assume that the language design will have inline procedures and that the compiler will eventually implement them. However, implementing a fast calling sequence has higher development priority than inline procedures because the benefits from a faster calling sequence are much greater.)

Most arguments will be passed in registers. Callers must save any live registers before calling a procedure (caller-saves). The stack pointer and the collector transaction queue are the only dedicated registers. The return address is passed as the 0th parameter in a register. Only move, add, and jump instructions will be used in the sequence.

Most procedures will have fixed-size stack frames, using the stack pointer directly as their frame pointer. Frames will not be explicitly linked together; instead the debugger and the exception handler trace back by mapping PC's to procedure descriptors that specify the size of the stack frame and the location of the return address.

Many "leaf" procedures won't even need a stack frame and won't do any memory references in the calling sequence. The few procedures that have dynamically sized frames will store their size in the frame, and their descriptor will indicate that. Nested procedures will be passed a pointer to the appropriate frame.

Tail-recursive calls will be optimized, saving a few more instructions. ("Tail recursive" traditionally refers to all calls in the tail position, recursive or not.)

The combination of argument passing in the registers, caller-saves, and tail recursion optimization will be especially effective for our styles of layered interfaces and object-oriented programming. Our programs have a large amount of "locality of arguments"; that is, the ith argument to a procedure is often passed on unchanged or only slightly changed as the ith argument to a sub-procedure that is often tail recursive.

The code generator must be written with the demands of the calling sequence in mind. It will try to target the evaluation of argument expressions into the appropriate registers, and it will reload values saved across procedure calls only on demand.

Raising exceptions with this sequence will be somewhat faster because RAISE will not have to examine register masks or restore saved registers as it traces back through the stack. Because of caller saves, a TRY handler is guaranteed that on entry all variable values are in known memory locations.

This calling sequence is significantly different from the VAX and the PRISM standards, both of which are much less efficient. Foreign code will be invoked either by a standard-call pragma for procedures or by generating stubs in the foreign-object-module converter.

16. Planning

The basic language tools, including the compiler, the binder, and much of the debugger, will have to be built from scratch. The past experiences of many different language projects show that, when requirements change dramatically, it is almost always better in the long run to build anew rather than hack an old

implementation. We don't want the design of our new environment to be needlessly constrained by the historical artifacts of 13-year-old Unix technology.

The Modula-2+ language must be frozen in its current state during the construction of the environment; tracking a moving language design greatly increases the cost of implementation. (And starting out with a freshly minted, "frozen" Modula-3 would be just as bad -- as with all language design, we would end up debugging the design after 9 months or so of experience.) Minor changes in the programmer-visible language, such as the simplification of the type system, will be allowed, but my definition of "minor" is very conservative.

Three people working full-time for 18 months should be able to implement the minimal set of tools needed to replace the current environment. A possible division of labor is:

> Editor, browser, pretty printer, definition-use database, integration with system modeler user interface.
>
> The AST interface and compiler. (I would sign up for this.)
>
> Binding, dynamic loading, shared libraries, the debugger.

Since tight integration with system modeling is an important goal of this proposal, careful planning with the system modeling people is crucial. However, implementation of the core tools (the compiler, the editor, and the runtime architecture) is not closely tied to having a working modeler until the later stages of the project. For example, the AST interface relies on having immutable file storage supplied by the modeler, but implementation and testing of the AST interface can use mutable Unix files.

The performance of pickles will probably have to be improved to reach its original goal of reading REF structures at disk speeds. But this proposal isn't closely tied to absolute pickle performance. Pickles already read and write large coarse-grained REF structures at near-disk speeds. The more pickles are improved, the finer the grain of structure we can use in the design.

This proposal relies heavily on REF structures, so we'll need the highly optimized reference-counting collector, including trace-and-sweep. The implementation of the collector need not be changed significantly in the new environment.

The OS people will have to provide copy-on-write for shared libraries, and provide some minimal support for managing the table of mapped-in libraries.

Someone in RPC land will have to rewrite "flume" using the AST interface.

The new environment will coexist with the current one for many months. Since the new environment will require very few changes to the operating system, the same machine can run programs from both environments at the same time. The lab can convert slowly at its own leisure. Taos and the Nub would probably be among last programs converted. On average, each member of the lab would probably spend a few weeks converting, but those weeks of effort would be distributed over a longer period.