Introduction

The Modular Programming System (MPS) is a facility for the development of large software systems. Our goal in creating it was a substantial improvement in the ease with which such systems are written, or rather the ease with which they evolve over a long period. We have not attempted much innovation in language features, since for the most part we were able to borrow satisfactory control and data handling facilities from Bliss [*1], Pascal [*2] and SUE [*3]; the few exceptions are discussed in the section on data structures below. Instead, we have concentrated on the more global parts of a programming system: control transfers between modules, binding of external references, specifying the execution context of a module. Following such diverse examples as assembly language and BBN Lisp [*4], we have also sought to define a logically complete virtual machine, every aspect of which is accessible to programs executing within it. The rest of this section outlines the essential features of MPS and comments on their precursors in other systems. ~~The body of the paper discusses these ideas in detail and explains what we have done to realize them.~~

Programming generality is perhaps the most important principle on which the system is based. MPS programs or computations are built out of modules. Each of these is a black box with a set of labelled ports emerging from it. The label on a port consists of a name, which is an arbitrary string, and a type. Modules can be used to build a computation by connecting ports which have compatible types. Programming generality requires that

(1) connections can be made without any changes to the modules;

(2) connections are not restricted by any properties of the modules except the port types;

(3) a configuration of modules with some connections established can be packaged, again without changes to the components, into a new module whose ports are the unconnected ports of the configuration, perhaps with new names. Such a configuration is functionally indistinguishable from an atomic module with the same ports. Of course this operation can be applied recursively.

An (atomic) module consists of code and context. The code consists of a body of instructions for the base machine on which MPS programs run, together with a symbol table and a context specifier which is a list of record types (as in Pascal [*2]; PL/I or Algol 68 call them structures). An example will explain this: in the Algol program

```
        begin integer a; b;
L:          begin Boolean c;
                ...
            end;
            ...
        end
```

the block labeled L would have the context specifier ((a: integer, b: integer), (c: Boolean)). The context consists of a list of record generators, each of which is (conceptually at least) a procedure which

returns a record whose type matches that of the specifier. The concepts of local variable and of record component are thus unified, and decisions about when new instances of variables are to be created can be programmed explicitly in the language. Readers familiar with Simula [*6] will recognize the ancestry of this idea. Of course it is possible to create any number of modules from the same code.

Control transfers between modules are handled by a uniform mechanism which subsumes the ideas of procedure call and return and coroutine linkage; these turn out to differ only in their conventions about when storage is allocated and released. Every control transfer passes a single argument record. The concepts of formal and actual argument list and of record are thus unified, so that all the machinery described below for handling records of variable type, for creating a record with default components, and for manipulating a record as a whole extends to parameter lists. It requires no special features to pass a variable number of arguments to a function, to return multiple result values or to intercept a function call and display the actual arguments. The syntax for ordinary procedure calls is unchanged, but it is now viewed as a convenient abbreviation for a sequence of more primitive actions: create an argument record, transfer control, receive control back, disassemble a result record.

When a module is created, its ports appear as variables in its context which are initialized to a special "unconnected" value. The code body contains a list of the ports, their types and the symbol table entries which contain their names. The process of connecting a port is called

binding; it is roughly analogous to the linking performed by a conventional loader. In MPS however, binding is entirely under control of the user. The creator of the module, or any one else who has access to it, can connect port P to port Q with the primitive bind (P,Q), or break the connector with bind (P, nil). This can be done at any time, whether or not execution has already begun or control been passed over the port. It is therefore possible to splice in tracing or advising procedures [*7: Warren], or to replace a module with a new version, without disturbing the state of a computation. There is also a set of standard binding procedures for binding things in standard ways, e.g. by matching names in a tree-structured naming hierarchy [*8: Multics].

An attempt to use an unbound port is translated into a call on a system primitive called the control fault handler; this idea is also taken from [*8]. The handler, however, takes no action of its own, but instead generates a called "control fault" signal. This is a standard control transfer preceded by a special mechanism which binds the port over which the transfer is going. The binding is done by following the chain of return links looking for one which lies in a scope which has enabled the signal. Associated with the enable is a catch-phrase, which declares a procedure to be called by the signal. The context of the catch-phrase includes the context of the scope in which it was enabled. The catch-phrase can return a value to the module which generated the signal, reject the signal and cause the binder to resume its search, or unwind control back to the module which enabled the signal. Other examples of signals might be "out of storage" or "end of file." An especially interesting example is "unwind" itself, which destroys each module which rejects the

signal; by enabling "unwind" a module can get control to clear itself up before being destroyed. Signals are a synthesis of PL/I ON-conditions [*9] and Snobol failure returns [*10], and provide a very attractive (and cheap) mechanism for handling unusual events. The way in which the binding is done means that modules which are not interested in a signal remain completely unaware of its existence,

New types of data structures are built up from primitive types such as integer (m...n) using the type constructors array, record, pointer, and union in the style of Algol 68 [*10] or Pascal [*2]. A type is described by a record of type "type descriptor" which can be manipulated by programs like any other record. Hence programs such as binders or debuggers can interpret type descriptors at run-time to check the validity of port connections or to display data structures. It is important to have only one copy of the source text for a type, no matter how many modules use it. To make this convenient, source programs can "include" declarations from other files. In addition a type descriptor carries information which identifies its source text, so it is possible to compare two types for structural or for literal identity. The major innovations in the type facility are a systematic treatment of generic functions, which subsumes operators like plus, assignment or subscripting as special cases, and complete user control over permissible coercions. This latter point is especially important for the binder, which needs to be able to insert coercions to obtain agreement of the types of two ports which it is trying to connect.

A final point, which has been mentioned several times above, is our

insistence that all of the data structures and procedures which form the infrastructure of MPS be accessible to programs. Type descriptors, symbol tables, frames allocated for local variables, ports and arguments are all MPS data structures whose type descriptors can be used to access and, when appropriate, modify them. The procedures which are invoked for binding, establishing context, allocating storage, signaling and passing control can all be replaced or advised. We have tried very hard not to provide any facilities which cannot be duplicated by ordinary MPS programs. As a result, most of the system facilities are in fact written as standard programs, as is the debugger.