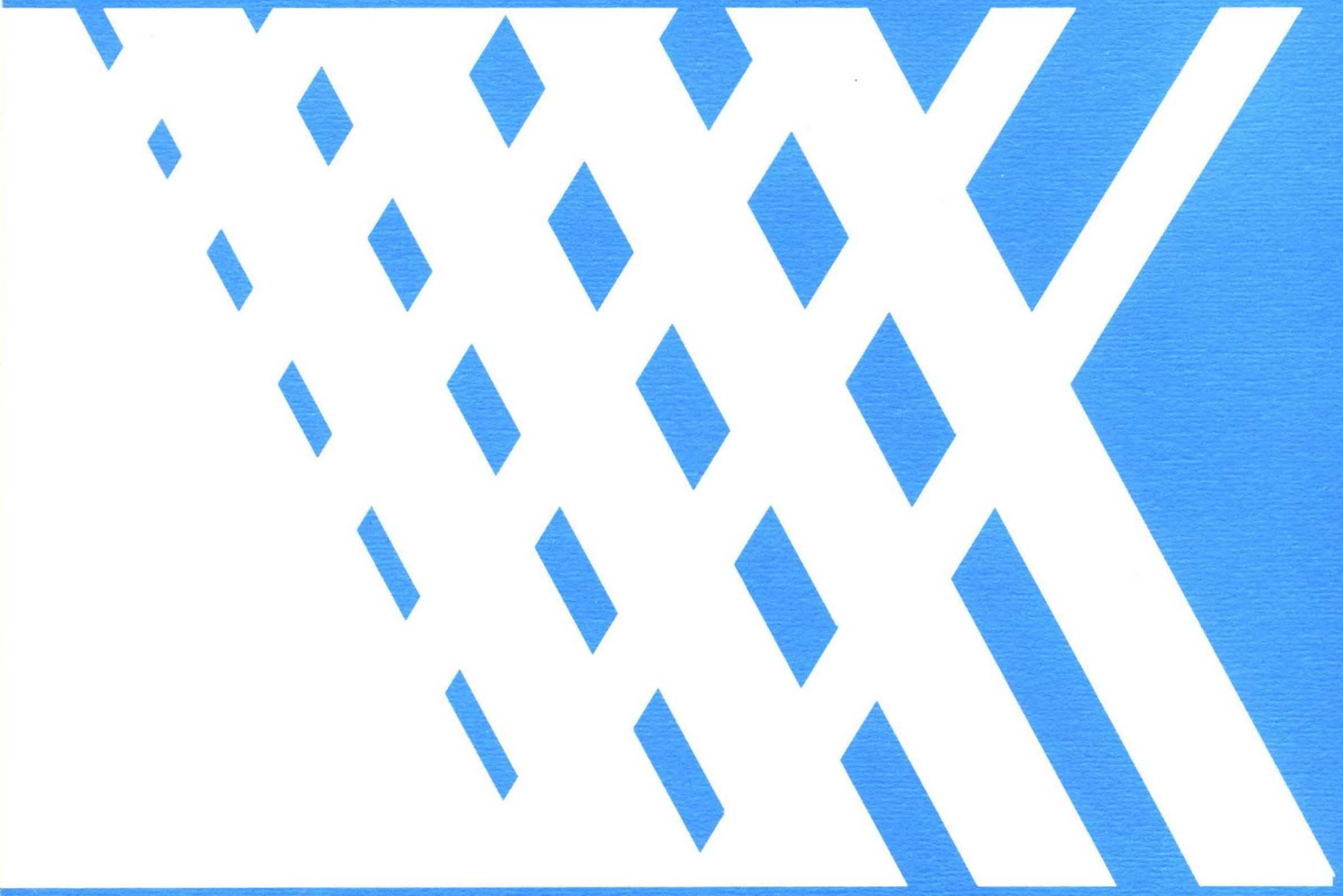
EARLY EXPERIENCE WITH MESA

CHARLES M. GESCHKE, JAMES H. MORRIS, JR., EDWIN H. SATTERTHWAITE



XEROX
PALO ALTO RESEARCH CENTER

Early Experience With Mesa

Charles M. Geschke, James H. Morris, Jr., Edwin H. Satterthwaite

CSL-76-6 October 1976

Abstract

The experiences of Mesa's first users -- primarily its implementers -- are discussed, and some implications for Mesa and similar programming languages are suggested. The specific topics addressed are:

- module structure and its use in defining abstractions,
- data-structuring facilities in Mesa,
- equivalence algorithm for types and type coercions,
- benefits of the type system and why it is breached occasionally,
- difficulty of making the treatment of variant records safe.

Key Words and Phrases

Programming languages, types, modules, data structures, systems programming

CR Categories: 4.22

XEROX

PALO ALTO RESEARCH CENTER

3333 Coyote Hill Road / Palo Alto / California 94304

Table of contents

Acknowledgements

The principal designers of Mesa, in addition to the authors, have been Butler Lampson and Jim Mitchell. The major portion of the Mesa operating system was programmed by Richard Johnsson and John Wick of the System Developement division of Xerox. In addition to those mentioned above, Douglas Clark, Howard Sturgis, and Niklaus Wirth have made helpful comments on earlier versions of this paper.

1. Introduction

What happens when professional programmers change over from an old-fashioned systems programming language to a new, modular, type-checked one like Mesa? Considering the large number of groups developing such languages, this is certainly a question of great interest.

This paper focuses on our experiences with strict type checking and modularization within the Mesa programming system. Most of the local structure of Mesa was inspired by, and is similar to, that of PASCAL [14] or ALGOL 68 [12], while the global structure is more like that of SIMULA 67 [1]. We have chosen features from these and related languages selectively, have cast them in a different syntax, and have added a few new ideas of our own. All this has been constrained by our need for a language to be used for the production of real system software right now. We believe that most of our observations are relevant to the languages mentioned above, and others like them, when used in a similar environment. We have therefore omitted a comprehensive description of Mesa and concentrated on annotated examples that should be intelligible to anyone familiar with a similar language. We hope that our experiences will help others who are creating or studying such languages.

An interested reader can find more information about the details of Mesa elsewhere. A previous paper [7] addresses issues concerning transfer of control. Another paper [3] discusses some more advanced data-structuring ideas. A paper on *schemes* [8] suggests another possible direction of advance. In this paper we shall restrain our desires to redesign or extend Mesa and simply describe how we are using the language as currently implemented.

The version of Mesa presented in this paper is one component of a continuing investigation into programming methodology and language design. Most major aspects of the language were frozen when implementation was begun in the autumn of 1974. Although we were dissatisfied with our understanding of certain design issues even then, we proceeded with implementation for the following reasons:

> We perceived a need for a "state of the art" implementation language within our laboratory. It seemed possible to combine some of our ideas into a design that was fairly conservative, but that would still dominate the existing and proposed alternatives.
>
> We wanted feedback from a community of users, both to evaluate those ideas that were ready for implementation and to focus subsequent research on problems actually encountered in building real systems.

We had accumulated a backlog of ideas about implementation techniques that we were anxious to try.

It is important to understand that we have consciously decided to attempt a complete programming system for demanding and sophisticated users. Their own research projects were known to involve the construction of "state of the art" programs, many of which tax the limits of available computing resources. These users are well aware of the capabilities of the underlying hardware, and they have developed a wide range of programming styles that they have been loath to abandon. Working in this environment has had the following consequences:

We could not afford to be too dogmatic. The language design is conservative and permissive; we have attempted to accommodate old methods of programming as well as new, even at some cost in elegance.

Efficiency is important. Mesa reflects the general properties of existing machines and contains no features that cannot be implemented efficiently (perhaps with some microcode assistance); for example, there is no automatic garbage collection.

A cross-compiler for Mesa became operational in the spring of 1975. We used it to build a small operating system and a display-oriented symbolic debugger. By early 1976, it was possible to run a system built entirely in Mesa on our target machine, and rewriting the compiler in its own language was completed in the summer of 1976. The basic system, debugger, and compiler consist of approximately 50,000 lines of Mesa code, the bulk of which was written by four people. Since mid-1976, the community of users and scope of application of Mesa have been expanding rapidly, but its most experienced and demanding users are still its implementers. It is in this context that we will try to describe our experiences and to suggest some tentative conclusions. Naturally, we have discovered some bugs and omissions in the design, and the implemented version of the language is already several years from the frontiers of research. We have tried to restrain our desire to redesign, however, and we report on Mesa as it is, not as we now wish it were.

The paper begins with a brief overview of Mesa's module structure. The uses of types and strict type checking in Mesa are then examined in some detail. The facilities for defining data structures are summarized, and an abstract description of the Mesa type calculus is presented. We discuss the rationale and methods for breaching the type system and illustrate them with a "type-strenuous" example that exploits several of the type system's interesting properties. A final section discusses the difficulties of handling variant records in a type-safe way.

2. Modules

Modules provide a capability for partitioning a large system into manageable units. They can be used to encapsulate *abstractions* and to provide a degree of *protection.* In the design of Mesa, we were particularly influenced by the work of Parnas [10], who proposes *information hiding* as the appropriate criterion for modular decomposition, and by the concerns of Morris [9] regarding protection in programming languages.

Module Structure

Viewed as a piece of source text, a *module* is similar to an ALGOL procedure declaration or a SIMULA class definition. It typically declares a collection of variables that provide a localized data base and a set of procedures performing operations upon that data base. Modules are designed to be compiled independently, but the declarations in one module can be made visible during the compilation of another by arranging to reference the first within the second by a mechanism called *inclusion.* To decouple the internal details of an implementation from its abstract behavior, Mesa provides two kinds of modules: *definitions* and *programs.*

A definitions module defines the interface to an abstraction. It typically declares some shared types and useful constants, and it defines the interface by naming a set of procedures and specifying their input/output types. Definitions modules claim no storage and have no existence at run-time. Included modules are usually definitions modules, but they need not be.

A program module provides the concrete *implementation* of an abstraction; it declares variables and specifies bodies of procedures. There can be a one-to-many relation between definitions modules and concrete implementations. At run-time, one or more instances of a module can be created, and a separate *frame* (activation record) is allocated for each. In this respect, module instances resemble SIMULA class objects. Unlike procedure instances, the lifetimes of module instances are not constrained to follow any particular discipline. Communication paths among modules are established dynamically as described below and are not constrained by, e.g., compile-time or run-time nesting relationships. Thus lifetimes and access paths are completely decoupled.

The following skeletal Mesa modules suggest the general form of a definitions module and one of its implementers:

```
Abstraction: DEFINITIONS =
    BEGIN
    ...
    it: TYPE = ... ;   rt: TYPE = ... ;
    ...
    p: PROCEDURE;
    p1: PROCEDURE [INTEGER];
    ...
    pi: PROCEDURE [it] RETURNS [rt];
    ...
    END

Implementer: PROGRAM IMPLEMENTING Abstraction =
    BEGIN OPEN Abstraction;
    x: INTEGER;
    ...
    p: PUBLIC PROCEDURE = <code for p>;
    p1: PUBLIC PROCEDURE [i: INTEGER] = <code for p1>;
    ...
    pi: PUBLIC PROCEDURE [x: it] RETURNS [y: rt] = <code for pi>;
    ...
    END
```

Longer but more complete and realistic examples can be found in the discussion of *ArrayStore* below; *ArrayStoreDefs* and *ArrayStore* correspond to *Abstraction* and *Implementer* respectively.

Mesa allows specification of attributes that can be used to control intermodular access to identifiers. In the definition of an abstraction, some types or record fields are of legitimate concern only to an implementer, but they involve or are components of other types that are parts of the advertised interface to the abstraction. Any identifier with the attribute PRIVATE is visible only in the module in which it is declared and in any module claiming to implement that module. Subject to the ordinary rules of scope, an identifier with the attribute PUBLIC is visible in any module that includes and *opens* the module in which it is declared. The PUBLIC attribute can be restricted by specifying the additional attribute READ-ONLY. By default, identifiers are PUBLIC in definitions modules and PRIVATE otherwise.

In the example above, *Abstraction* contains definitions of shared types and enumerates the elements of a procedural interface. *Implementer* uses those type definitions and provides the bodies of the procedures; the compiler will check that an actual procedure with the same name and type is supplied for each public procedure declared in *Abstraction.*

A module that uses an abstraction is called a *client* of that abstraction. Interface definitions are obtained by including the *Abstraction* module. Any instance of a client must be connected to an instance of an appropriate implementer before the actual operations of the abstraction become available. This connection is called *binding*, and there are several ways to do it.

Binding Mechanisms

When a relatively static and purely procedural interface between modules is acceptable, the connection can be made in a conventional way. Consider the following skeleton:

```
Client1: PROGRAM =
    BEGIN OPEN Abstraction;
    ...
    px: EXTERNAL PROCEDURE;
    ...
    p[]; px[];
    ...
    END.
```

A client module can request a system facility called the *binder* to locate and assign appropriate values to all external procedure names, such as *px*. The binder follows a well-defined *binding path* from module instance to module instance. When the binder encounters an actual procedure with the same name as, and a type compatible with, an external procedure, it makes the linkage. The compiler automatically inserts an EXTERNAL procedure declaration for any procedure identifier, such as *p*, that is mentioned by a client but defined only in an included definitions module. The binder also checks that all identifiers from a single definitions module are bound consistently (i.e. to a single implementer).

The observant reader will have noticed that this binding mechanism and the undisciplined lifetimes of module instances leave Mesa programs vulnerable to dangling reference problems. We are not happy about this, but so far we have not observed any serious bugs attributable to such references.

As an alternate binding mechanism, Mesa supports the SIMULA paradigm as suggested by the following skeleton (which assumes that *x* is a public variable):

```
Client2: PROGRAM =
    BEGIN OPEN Abstraction;
    frame: POINTER TO FRAME[Implementer] ← NEW Implementer;
    ...
    frame↑.x ← 0;
    frame↑.p[];
    ...
    END.
```

Here, the client creates an instance of *Implementer* directly. Through a pointer to the frame of that instance, the client can access any public variable or invoke any public procedure. Note that the relevant declarations are in *Implementer;* the *Abstraction* module is included only for type definitions. Some of the binding has been moved to compile-time. In return for a wider, not necessarily procedural interface (and potentially more efficient code), the client has committed himself to using a particular implementation of the abstraction.

Because Mesa has procedure variables it is possible for a user to create any binding regime he wishes simply by writing a program that distributes procedures. Some users have created their own version of SIMULA classes. They have not used the binding mechanism described above for a number of reasons. First, the actual implementation of an abstract object is sometimes unknown when a program is compiled or instantiated; there might be several coexisting implementations, or the actual implementation of a particular object might change dynamically. Their binding scheme deals with such situations by representing objects as record structures with procedure-valued fields. The basic idea was described in connection with the implementation of streams in OS6 [11]: some fields of each record contain the state information necessary to characterize the object, while others contain procedure values that implement the set of operations. If the number of objects is much larger than the number of implementations, it is space-efficient to replace the procedure fields in each object with a link to a separate record containing the set of values appropriate to a particular implementation. When this binding mechanism is used, interface specifications consist primarily of type definitions as suggested by the following skeleton:

```
ObjectAbstraction: DEFINITIONS =
    BEGIN
    Handle: TYPE = POINTER TO Object;
    Object: TYPE = RECORD [
        ops: POINTER TO Operations,
        state: POINTER TO ObjectRecord,
        ...];
    Operations: TYPE = RECORD [p1: PROCEDURE [Handle, INTEGER], ...];
    END.
```

A client invokes a typical operation by writing *handle*↑.*ops*↑.*p1*[*handle*, *x*] where *handle* is of type *Handle*.

Observations

We believe that we could not have built the current Mesa system if we had been forced to work with large, logically monolithic programs. Assembly language programmers are well aware of the benefits of modularity, but many designers of high level programming languages pay little attention to the problems of independent compilation and instantiation. Since these capabilities will be grafted on anyway, they should be anticipated in the original design. We have more to say about interface control in our discussion of types, but it is hard to overestimate the value of articulating abstractions, centralizing their definitions, and propagating them through the inclusion mechanism.

3. The Mesa Type System

Strict vs. non-strict type checking

A widely held view is that the purpose of type declarations is to allow one to write more succinct programs. For example, the ALGOL 60 declarations

real *x,y;* **integer** *i,j;*

allow one to attach two different interpretations to the symbol "+" in the expressions *x+y* and *i+j*. Similarly, the declaration

x: RECORD[*a:* [0..7], *b:* [0..255]]

permits one to write *x.a* and *x.b* in place of descriptions of the shifting and masking that must occur. Descriptive declarations also allow utility programs such as debuggers to display values of variables in a helpful way when the type is not encoded as part of the value.

This view predominated in an earlier version of Mesa. Type declarations were used primarily as devices to improve the expressive power and readability of the language. Types were ignored by the compiler except to discover the number of bits involved in an operation. In contrast, the current version of Mesa checks type agreement as rigorously as languages such as PASCAL or ALGOL 68, potentially rendering compile-time complaints in great volume. This means in effect that the language is more redundant, since there are fewer programs acceptable to the compiler.

What benefit do we hope to gain by stricter checking and the attendant obligations on the programmer? We expect that imposing additional structure on the data space of the program and checking it mechanically will make the modification and maintenance of programs easier. The type system allows us to write down certain design decisions. The type checker is a tool that is used to discover violations of the conventions implied by those decisions without a great expenditure of thought.

Type Expressions

Mesa provides a fairly conventional set of expressions for describing types; detailed discussions of the more important constructors are available elsewhere [3]. We shall attempt just enough of an introduction to help in reading the subsequent examples and concentrate upon the relations among types.

There is a set of predefined basic types and a set of *type operators* which construct new types. The arguments of these operators may be other types, integer constants, or identifiers with no *a priori* meanings. Most of the operators are familiar from languages such as PASCAL or ALGOL 68, and the following summary emphasizes only the differences.

Basic Types

The basic types are INTEGER, BOOLEAN, CHARACTER, and UNSPECIFIED, the last of which is a one-word, wild card type.

Enumerated Types

If a_1, a_2, ..., a_n are distinct identifiers, the form $\{a_1, a_2, ..., a_n\}$ denotes an ordered type of which the identifiers constantly denote the allowed values.

Unique Types

If *n* is a manifest (compile-time) constant of type INTEGER, the form UNIQUE[*n*] denotes a type distinct from any other type. The value of *n* determines the amount of storage allocated for values of that type, which are otherwise uninterpreted. Its use is illustrated by the *ArrayStore* example in Section 4.

Record Types

If T_1, T_2, ... T_n are types and f_1, ..., f_n are distinct identifiers, then the form RECORD[f_1: T_1, f_2: T_2, ..., f_n: T_n] denotes a record type. The f_i are called *field selectors.* As usual, the field selectors are used to access individual components; in addition, linguistic forms called *constructors* and *extractors* are available for synthesizing and decomposing entire records. The latter forms allow either keyword notation, using the field names, or positional notation. Intermodule access to individual fields can be controlled by specifying the attributes PUBLIC, PRIVATE, or READ-ONLY; if no such attributes appear, they are inherited from the enclosing declaration. Some examples:

```
Thing: TYPE = RECORD [n: INTEGER, p: BOOLEAN];
v: Thing;  i: INTEGER;  b: BOOLEAN;
...
IF v.p THEN v.n ← v.n + 1;    -- field selection
v ← [100, TRUE];              -- a positional constructor
v ← [p:b, n:i];               -- a keyword constructor
[n:i, p:b] ← v;               -- the inverse extractor.
```

Pointer Types

If T is a type, the form POINTER TO T denotes a pointer type. If *x* is a variable of that type, then *x*↑ *dereferences* the pointer and designates the object pointed to, as in PASCAL. If *v* is of type T then @*v* is its address with type POINTER TO T. The form POINTER TO READ-ONLY T denotes a similar type; however, values of this type cannot be used to change the indirectly referenced object. Such pointer types were introduced so that objects could be passed by reference across module interfaces with assurance that their values would not be modified.

Array Types

If T_i and T_c are types, the form ARRAY T_i OF T_c denotes an array type. T_i must be a finite ordered type. An array *a* maps an index *i* from the index type T_i into a value *a*[*i*] of the component type T_c. If *a* is a variable, the mapping can be changed by assignment to *a*[*i*].

Array Descriptor Types

If T_i and T_c are types, the form DESCRIPTOR FOR ARRAY T_i OF T_c denotes an array descriptor type. T_i must be an ordered type. An array descriptor value provides indirect access to an array and contains enough auxiliary information to determine the allowable indices as a subrange of T_i.

Set Types

If T is a type, the form SET OF T denotes a type, values of which are the subsets of the set of values of T. T must evaluate to an enumerated type.

Transfer Types

If T_1, ... T_i, T_j, ... T_n are types and f_1, ..., f_i, f_j, ..., f_n are distinct identifiers, then the form PROCEDURE [f_1: T_1, ..., f_i: T_i] RETURNS [f_j: T_j, ... f_n: T_n] denotes a procedure type. Each non-local control transfer passes an argument record; the field lists enclosed by the paired brackets, if not empty, implicitly declare the types of the records accepted and returned by the procedure [7]. If *x* has some transfer type, a control transfer is invoked by the evaluation of *x*[e_1, ..., e_i], where the bracketed expressions are used to construct the input record, and the value is the record constructed in preparation for the transfer that returns control.

The symbol PROCEDURE can be replaced by several alternatives that specify different transfer disciplines with respect to name binding, storage allocation, etc., but the argument transmission mechanism is uniform. Transfer types are full-fledged types; it is possible to declare procedure variables and otherwise to manipulate procedure values, which are represented by procedure descriptors. Indeed, some of the intermodule binding mechanisms described previously depend crucially upon the assignment of values to procedure variables.

Subrange Types

If T is INTEGER or an enumerated type, and *m* and *n* are manifest constants of that type, the form T[*m*..*n*] denotes a finite, ordered subrange type for which any legal value *x* satisfies $m \leq x \leq n$. If T is INTEGER, the abbreviated form [*m*..*n*] is accepted. These types are especially useful as the index types of arrays. Other notational forms, e.g. [*m*..*n*), allow intervals to be open or closed at either endpoint.

Finally, Mesa has adapted PASCAL's variant record concept to provide values whose complete type can only be known after a run-time discrimination. Because they are of more than passing interest, variant records are discussed separately in Section 5.

Declarations and Definitions

The form

v: Thing ← e

declares a variable *v* of type *Thing* and initializes it to the value of *e;* the form

v: Thing = e

is similar except that *v* cannot be assigned to subsequently. When *e* itself is a manifest constant, this form makes *v* into such a constant also.

This syntax is used for the introduction of new type names, using the special type TYPE. Thus

Thing: TYPE = *TypeExpression*

introduces *Thing.* This approach came from ECL [13], in which a type is a value that can be computed by a running program and then used to declare variables. In Mesa, however, *TypeExpression* must be constant.

Recursive type declarations are essential for describing most list structures and are allowed more generally whenever they make sense. To accommodate mutually recursive list structure, forward references to type identifiers are allowed and do not yield "uninitialized" values. (This is to be contrasted with forward references to ordinary variables.) In effect, all type expressions within a scope are evaluated simultaneously. Meaningful recursion in a type declaration usually involves the type constructor POINTER; in corresponding values, the recursion involves a level of indirection and can be terminated by the empty pointer value NIL. Recursion that is patently meaningless is rejected by the compiler; for example

```
r: TYPE = RECORD [left, right: r]  -- not permitted
a: TYPE = ARRAY [0..10) OF s;
s: TYPE = RECORD [i: INTEGER, m: a]  -- not permitted.
```

Similar pathological types have been noted and prohibited in ALGOL 68 [6].

Equivalence of Type Expressions

One might expect that two identical type expressions appearing in different places in the program text would always stand for the same type. In ALGOL 68 they do. In Mesa (and certain implementations of PASCAL) they do not. Specifically, the type operators RECORD, UNIQUE, and {...} generate new types whenever they appear in the text.

The original reasons for this choice are not very important, but we have not regretted the following consequences for records:

All modules wishing to communicate using a shared record type must obtain the definition of that type from the same source. In practice, this means that all definitions of an abstraction tend to come from a single module; there is less temptation to declare scattered, partial interface definitions.

Tests for record type equivalence are cheap. In our experience, most record types contain references to other record types, and this linking continues to a considerable depth. A recursive definition of equivalence would, in the worst case, require examining many modules unknown and perhaps unavailable to the casual user of a record type or, alternatively, copying all type definitions supporting a particular type into the symbol table of any module mentioning that type.

The rule for record equivalence provides a mechanism for *sealing* values that are distributed to clients as passkeys for later transactions with an implementer. Suppose that the following declaration occurs in a definitions module:

Handle: PUBLIC TYPE = RECORD [*value:* PRIVATE *Thing*].

The PRIVATE attribute of *value* is overridden in any implementer of *Handle*. A client of that implementer can declare variables of type *Handle* and can store or duplicate values of that type. However, there is no way for the client to construct a counterfeit *Handle* without violating the type system. Such sealed types appear to provide a basis for a compile-time capability scheme [2].

Finally, this choice has not caused discomfort because programmers are naturally inclined to introduce names for record types anyway.

The case for distinctness of enumerated types is much weaker; we solved the problem of the exact relationships among such types as {*a, b, c*}, {*c, b, a*}, {*a, c*}, {*aa, b, cc*}, etc., by specifying that all these types are distinct. In this case, we are less happy that identical sequences of symbols construct different enumerated types.

Why did we not choose a similar policy for other types? It would mean that a new type identifier would have to be introduced for virtually every type expression, and we found it to be too tedious. In the case of procedures we went even further in liberalizing the notion of equivalence. Even though the formal argument and result lists are considered to be record declarations, we not only permit recursive matching but also ignore the field selectors in doing the match. We were unwilling to abandon the idea that procedures are mappings in which the identifiers of bound variables are irrelevant. We also had a pragmatic motivation. In contrast to records, where the type definitions cross interface boundaries, procedural communication among modules is based upon procedure values, not procedure types. Declaring named types for all interface procedures seemed tiresome. Fortunately, all argument records are constructed in a standard way, so this view causes no implementation problems.

To summarize, we state an informal algorithm for testing for type equivalence. Given one or more program texts and two particular type expressions in them:

1. Tag each occurrence of RECORD, UNIQUE, and {...} with a distinct number.

2. Erase all the variable names in formal parameter and result lists of procedures.

3. Proceed to compare the two expressions, replacing type identifiers with their defining expressions whenever they are encountered. If a difference (possibly in a tag attached in step 1) is ever encountered, the two type expressions are not equivalent. Otherwise they are equivalent.

The final step appears to be a semi-decision procedure since the existence of recursive types makes it impossible to eliminate all the identifiers. In fact, it is always possible to tell when one has explored enough (cf. [5], Section 2.3.5, Exercise 11).

Coercions

To increase the flexibility of the type system Mesa permits a variety of implicit type conversions beyond those implied by type equivalence. They fall into two categories: *free coercions* and *computed coercions.*

Free Coercions

Free coercions involve no computation whatsoever. For two types T and S we write T $\subseteq$ S if any value of type T can be stored into a variable of type S without checking, change of representation, or other computation. (By "store" we mean to encompass assignment, parameter passing, result passing, and all other value transmission.) The following recursive rules show how to compute the relation $\subseteq$, assuming that equivalence has already been accounted for.

1. T $\subseteq$ T.

In the following assume that T $\subseteq$ S.

2. T[*i..j*] $\subseteq$ S if i is the minimum value of type S.
   The restriction is necessary because we chose to represent values of a subrange type relative to its minimum value. Coercions in other cases require computation. Similarly,

3. T[*i..j*] $\subseteq$ S[*i..k*] iff $j \leq k$.

4. *var* T $\subseteq$ S if *var* is a variant of T (cf. Section 5).

5. RECORD[*f:* T] $\subseteq$ S for any field name *f* unless *f* has the PRIVATE attribute.

6. POINTER TO T $\subseteq$ POINTER TO READ-ONLY S.
   In other words, one can always treat a pointer as a read-only pointer, but not vice versa.

7. POINTER TO READ-ONLY T ⊆ POINTER TO READ-ONLY S.
   The relation POINTER TO T ⊆ POINTER TO S is *not* true because it would allow

   ```
   ps: POINTER TO S;
   pt: POINTER TO T = @t;
   ps ← pt;
   ps↑ ← s;
   ```

   which is a sneaky way of accomplishing "$t \leftarrow s$," which is not allowed unless S ⊆ T.

8. ARRAY I OF T ⊆ ARRAY I OF S.
   Note that the index sets must be the same.

9. PROCEDURE [S'] RETURNS [T] ⊆ PROCEDURE [T'] RETURNS [S] if T' ⊆ S' as well. Here the relation between the input types is the reverse of what one might expect.

Subrange Coercions

Coercions between subranges require further comment. As others have noted [4], associating range restrictions with types instead of specific variables leads to certain conceptual problems; however, we wanted to be able to fold range restrictions into more complex constructed types. We were somewhat surprised by the subtlety of this problem, and our initial solutions allowed several unintended breaches of the type system.

Values of an ordered type and all its subranges are interassignable even if they do not satisfy case (2) or (3) from above, and this is an example of a computed coercion. Code is generated to check that the value is in the proper subrange and to convert its representation if necessary. It is important to realize that the relation of computed coercability cannot be extended recursively as was done above. Consider the declarations

```
x: [0..100] ← 15;
y: [10..20];
px: POINTER TO READ-ONLY [0..100] ← @x;
py: POINTER TO READ-ONLY [10..20];
```

The assignment $y \leftarrow x$ is permitted because x is 15; 5 is stored in y since is value its represented relative to 10. However, the assignment $py \leftarrow px$, which rule 7 might suggest, is not permitted because the value of x can change and there is no reasonable way to generate checking code. Even if the value of x cannot change, we could not perform any change in representation because the value 15 is shared. Similar problems arise when one considers rules 6, 8, and 9.

Other Computed Coercions

Research in programming language design has continued in parallel with our implementation work, and some proposals for dealing with uniform references [3] and generalizations of classes [8] suggested adding the following computed coercions to the language:

Dereferencing: POINTER TO $T \rightarrow T$
Deproceduring: PROCEDURE RETURNS $T \rightarrow T$
Referencing: $T \rightarrow$ POINTER TO T.

Initially we had intended to support contextually implied application of these coercions much as does ALGOL 68. Reactions of Mesa's early users to this proposal ranged from lukewarm to strongly negative. In addition, the data structures and accounting algorithms necessary to deduce the required coercions and detect pathological types substantially complicated the compiler. We therefore decided to reconsider our decision even after the design and some of the implementation had been done. The current language allows subrange coercion as described above. There is no uniform support for other computed coercions, but automatic dereferencing is invoked by the operators for field extraction and array indexing. Thus such forms as $p\uparrow.f$ and $a\uparrow\uparrow[i]$, which are common when indirection is used extensively, may be written as $p.f$ and $a[i]$.

There are hints of a significant problem for language designers here. Competent and experienced programmers seem to believe that coercion rules make their programs less understandable and thus less reliable and efficient. On the other hand, techniques being developed with the goal of decreasing the cost of creating and changing programs seem to build heavily upon coercion. Our experience suggests that such work should proceed with caution.

Why is coercion distrusted? Our discussions with programmers suggest that the reasons include the following:

> Mesa programmers are familiar with the underlying hardware and want to be aware of the exact consequences of what they write.
>
> Many of them have been burned by forgotten indirect bits and the like in previous programming and are suspicious of any unexpected potential for side effects.
>
> To some extent, coercion negates the advantages of type checking. One view of coercion is that it corrects common type errors, and some of the detection capability is sacrificed to obtain the correction.

We conjecture that the first two objections will diminish as programmers learn to think in terms of higher-level abstractions and to use the type checking to advantage.

The third objection appears to have some merit. We know of no system of coercions in which strict type checking can be trusted to flag all coercion errors, and such errors are likely to be especially subtle and persistent. The difficulties seem to arise from the interactions of coercion with generic operators. In ALGOL 68, there are rules about

"loosely related" types that are intended to avoid this problem, but the identity operators still suffer. With the coercion rules that had been proposed for Mesa, the following trap occurs. Given the declaration *p, q:* POINTER TO INTEGER, the Mesa expressions $p\uparrow = q\uparrow$ and $2*p = 2*q$ would compare integers and give identical results; on the other hand, the expression $p = q$ would compare pointers and could give a quite different answer. In the presence of such traps, we believe that most programmers would resolve to supply the "↑" always. If this is their philosophy, coercions can only hide errors. Even if such potentially ambiguous expressions as $p = q$ were disallowed, this example suggests that using coercion to achieve representational independence can easily destroy referential transparency instead.

4. Experiences with strict type checking

It is hard to give objective evidence that increasing compile-time checking has materially helped the programming process. We believe that it will take more effort to get one's program to compile and that some of the effort eliminates errors that would have shown up during testing or later, but the magnitude of these effects is hard to measure. All we can present at the moment are testimonials and anecdotes.

A testimonial

Programmers whose previous experience was with unchecked languages report that the usual fear and trepidation that accompanied making modifications to programs has substantially diminished. Under previous regimes they would never change the number or types of arguments that a procedure took for fear that they would forget to fix all of the calls on that procedure. Now they know that all references will be checked before they try to run the program.

An anecdote

The following kind of record is used extensively in the compiler

RelativePtr: TYPE = [0..37777_8];
TaggedPtr: TYPE = RECORD[*tag:* $\{t_0,t_1,t_2,t_3\}$, *ptr: RelativePtr*].

This record consists of a two-bit tag and a 14-bit pointer. As an accident of the compiler's choice of representation, the expressions x and *TaggedPtr*[t_0,x] generated the same internal value. The non-strict type checker considered these types equivalent, and unwittingly we used *TaggedPtr*s in many places actually requiring *RelativePtr*s. As it happened, the tag in these contexts was always t_0.

The compiler was working well, but one day we made the unfortunate decision to redefine *TaggedPtr* to be

RECORD[*ptr: RelativePtr, tag:* $\{t_0,t_1,t_2,t_3\}$]

This caused a complete breakdown, and we hastily unmade that decision because we were unsure about what parts of the code were unintentionally depending upon the old representation. Later, when we submitted a transliteration of the compiler to the strict type checker we found all the places where this error had been committed. Nowadays, making such a change is routine. In general, we believe that the benefits of static checking are significant and cost-effective once the programmer learns how to use the type system effectively.

A shortcoming

The type system is very good at detecting the difference in usage between T and POINTER TO T; however, programmers often use array indices as pointers, especially when they want to perform arithmetic on them. The difference between an integer used as a pointer and an integer used otherwise is invisible to the type checker. For example, the declaration

map: ARRAY [*i..j*] OF INTEGER[*m..n*];

defines a variable *map* with the property that compile-time type checking cannot distinguish between legitimate uses of *k* and *map*[*k*]. Furthermore, if $m \leq i$ and $j \leq n$, even a run-time bounds check could never detect a use of *k* when *map*[*k*] was intended. We have observed several troublesome bugs of this nature and would like to change the language so that indices of different arrays can be made into distinct types.

Violating the type system

One of the questions often asked about languages with compile-time type checking is whether it is possible to write real programs without violating the type system. It goes without saying that one can bring virtually any program within the confines of a type system by methods analogous to the silly methods for eliminating goto's; e.g., simulate things with integers. However, our experience has been that it is not always desirable to remain within the system, given the realities of programming and the restrictiveness of the current language. There are three reasons for which we found it desirable to evade the current type system.

Sometimes the violation is logically necessary. Fairly often one chooses to implement part of a language's run-time system in the language itself. There are certain things of this nature that cannot be done in a type-safe way in Mesa, or any other strictly type-checked language we know. For example, the part of the system that takes the compiler's output and creates values of type PROCEDURE must exercise a rather profound loophole in turning data into program. Another example, discussed in detail below, is a storage allocator. Most languages with compile-time checking submerge these activities into the implementation and thereby avoid the need for type breaches.

Sometimes efficiency is more important than type safety. In many cases the way to avoid a type breach is to redesign a data structure in a way that takes more space, usually by introducing extra levels of pointers. The section on variant records gives an example.

Sometimes a breach is advisable to increase type checking elsewhere. Occasionally a breach could be avoided by declaring two distinct types to be the same but merging them would reduce a great deal of checking elsewhere. The *ArrayStore* example below illustrates this point.

Given these considerations, we chose to allow occasional breaches of the type system, making them as explicit as possible. The advantages of doing this are two-fold. First, making breaches explicit makes them less dangerous since they are clearer to the reader. Second, their occurrences provide valuable hints to a language designer about where the type system needs improvement.

One of the simplest ways to breach the Mesa type system is to declare something to be UNSPECIFIED. The type checking algorithm regards this as a one-word, don't-care type that matches any other one word type. This is similar to PL/1's UNSPEC. We have come to the conclusion that using UNSPECIFIED is too drastic in most cases. One usually wants to turn off type checking in only a few places involving a particular variable, not

everywhere. In practice there is a tendency to use UNSPECIFIED in the worst possible way: at the interfaces of modules. The effect is to turn off type checking in other peoples' modules without their knowing it!

As an alternative, Mesa provides a general type transfer function, RECAST, that (without performing any computation) converts between any two types of equal size. It can often be used instead of UNSPECIFIED. In cases where we had declared a particular variable UNSPECIFIED, we now prefer to give it some specific type and to use RECAST whenever it is being treated in a way that violates the assumptions about that type.

The existence of RECAST makes many decisions much less painful. Consider the type CHARACTER. On the one hand we would like it to be disjoint from INTEGER so that simple mistakes would be caught by the type checker. On the other hand, one occasionally needs to do arithmetic on characters. We chose to make CHARACTER a distinct type and use RECAST in those places where character arithmetic is needed. Why reduce the quality of type checking everywhere just to accommodate a rare case?

Pointer arithmetic is a popular pastime for system programmers. Rather than outlawing it, or even requiring a RECAST, Mesa permits it in a restricted form. One can add or subtract an integer from a pointer to produce a pointer of the same type. One can subtract two pointers of the same type to produce an integer. The need for more exotic arithmetic has not been observed.

Here is a typical example: it is common to use a large contiguous area of memory to hold a data structure consisting of many records; e.g., a parse tree. To conserve space one would like to make all pointers relative to the start of the area, thus reducing the size of pointers that are internal to the structure. Furthermore, one might like to move the entire area, possibly via secondary storage. These needs would be met by an unimplemented feature called the *tied pointer*. The idea is that a certain type of pointer would be made relative to a designated base value and this value would be added just before dereferencing the pointer. In other words, if *ptr* were declared to be tied to *base* then *ptr*↑ actually would mean (*base*+*ptr*)↑. Since tied pointers have not yet been implemented, this notation is in fact used extensively within the Mesa compiler. Subsequent versions of Mesa will include tied pointers, and this temporary loophole will be reconsidered.

The Skeleton Type System

Once we provided the opportunity for evading the official type system, we had to ask ourselves just why we thought certain breaches were safe while others were not. Ultimately, we came to the conclusion that the only really dangerous breaches of the type system were those that require detailed knowledge of the run-time environment. First and foremost, fabricating a procedure value requires a detailed understanding of how various structures in memory are arranged. Second, pointer types also depend on various memory structures being set up properly and should not be passed through loopholes without some care. In contrast, the distinction between the two types RECORD [*a,b:* INTEGER] and RECORD[*c,d:* INTEGER] is not vital to the run-time system's integrity. To be sure, the user might wish to keep them distinct, but using a loophole to store one into the other would go entirely unnoticed by the system.

The present scheme that is used to judge the appropriateness of RECAST transformations merely checks to ensure that the source and destination types occupy the same number of bits. Since most of the code invoking RECAST has been written by Mesa implementers, this simplified check has proved to be sufficient. However, as the community of users has grown, we have observed a justifiable anxiety over the use of RECAST. Users fear that unchecked use of this escape will cause a violation of some system convention unknown to them.

We are in the process of investigating a more complete and formal skeletal type system that will reduce the hazards of the present RECAST mechanism. Its aim is to ensure that although a RECAST may do great violence to user-defined type conventions, the system's type integrity will not be violated.

Example -- A compacting storage allocator

A module that provides many arrays of various sizes by parceling out pieces of one large array is an interesting benchmark for a systems programming language for a number of reasons:

a. It taxes the type system severely. We must deal with an array containing variable length, heterogeneous objects, something one can't declare in Mesa.

b. The clients of the allocator wish to use it for arrays of differing types. This is a familiar polymorphism problem.

c. As a programming exercise, the module can involve intricate pointer manipulations. We would like help to prevent programming errors such as the ubiquitous address/contents confusion.

d. A nasty kind of bug associated with the use of such packages is the so-called dangling reference problem: someone might use some space after he has relinquished it.

e. Another usage bug, peculiar to compacting allocators, is that a client might retain a pointer to storage that the compacter might move.

The first two problems make it impossible to stay entirely within the type system. One's first impulse is to declare everything unspecified and proceed to program as in days of yore. The remaining problems are real ones, however, and we are reluctant to turn off the entire type system just when we need it most. The following is a compromise solution:

To deal with problem (a) we shall have two different ways of talking about the array to be parceled out, which we shall call *Storage*. From a client's point of view the storage is accessible through the definitions shown in the module *ArrayStoreDefs*. (cf. Figure 1)

```
ArrayStoreDefs: DEFINITIONS =
  BEGIN
  ArrayPtr: TYPE = POINTER TO PR;
  PR: TYPE = POINTER TO R;
  R: TYPE = RECORD [ p: Prefix,
                     a: ARRAY [0..0] OF Thing ];
  Prefix: TYPE = RECORD [ backp: PRIVATE ArrayPtr,
                          length: READ-ONLY INTEGER ];
  Thing: TYPE = UNIQUE[16];
  AllocArray: PROCEDURE [length: INTEGER]
                RETURNS [new: ArrayPtr];
  FreeArray: PROCEDURE [dying: ArrayPtr];
  END
```

Figure 1. Definitions Module

These definitions suggest that the client can get *ArrayPtrs* (i.e. pointers to pointers to array records) by calling *AllocArray* and can relinquish them by calling *FreeArray*. The PRIVATE attribute on *backp* means that he cannot access that field at all. The READ-ONLY attribute on *length* means that he cannot change it. Of course these restrictions do not apply to the implementing module. The type *Thing* occupies 16 bits of storage (one word) and matches no other type. Intuitively, it is our way of faking a type variable. The implementing module *ArrayStore* is shown in Figure 2. It declares the array *Storage* to create the raw material for allocation. We chose to declare its element type UNSPECIFIED. This means that every transaction involving *Storage* is an implicit invocation of a loophole. Specifically the initializations of *beginStorage* and *endStorage* store pointers to UNSPECIFIED into variables declared as pointers to *R*.

The general representation scheme is as follows: the storage area [*beginStorage..nextR*) consists of zero or more *R*s, each with the form $\langle backp, length, e_0, \ldots, e_{(length-1)} \rangle$, where *length* varies from sequence to sequence. The array represented by the record is $\langle e_0, \ldots, e_{(length-1)} \rangle$. If *backp* is not NIL then *backp* is an address in *Table* and *backp*↑ is the address of *backp* itself. If *Table*[*i*] is not NIL, it is the address of one of these records. (cf. Figure 3)

After the initialization, *Storage* is not mentioned again. All the subsequent type breaches in *ArrayStore* are of the pointer arithmetic variety. The expression *endStorage-nextR* in *AllocArray* subtracts two *PR's* to produce an integer. The type checker is not entirely asleep here: if we slipped up and wrote

```
IF n+ovh > endStorage-n
```

there would be a complaint, because the left hand side of the comparison is an integer and the right is a *PR*. The assignment

```
nextR ← nextR+(n+ovh)
```

at the end of *AllocArray* also uses the pointer arithmetic breach. The rule *PR*+INTEGER = *PR* makes sense here because *n+ovh* is just the right amount to add to *nextR* to produce the next place where an *R* can go.

```
DIRECTORY ArrayStoreDefs: FROM "ArrayStoreDefs";
DEFINITIONS FROM ArrayStoreDefs;

ArrayStore: PROGRAM IMPLEMENTING ArrayStoreDefs =
  BEGIN
  Storage: ARRAY [0..StorageSize) OF UNSPECIFIED;
  StorageSize : INTEGER = 2000;
  Table: ARRAY TableIndex OF PR;
  TableIndex: TYPE = [0..TableSize);
  TableSize: INTEGER = 500;
  beginStorage: PR = @Storage[0];
          -- the address of Storage[0]
  endStorage: PR = @Storage[StorageSizc];
  nextR: PR ← beginStorage;  -- next space to put an R
  beginTable: ArrayPtr = @Table[0];
  endTable: ArrayPtr = @Table[TableSize];
  ovh: INTEGER = SIZE[Prefix];  -- overhead

  AllocArray: PUBLIC PROCEDURE [n: INTEGER]
                    RETURNS [new: ArrayPtr] =
    BEGIN i:TableIndex;
    IF n<0 OR n>77777B-ovh THEN ERROR;
    IF n+ovh > endStorage-nextR THEN
      BEGIN
      Compact[];
      IF n+ovh > endStorage-nextR THEN ERROR;
      END;
    -- Find a table entry
    FOR i IN TableIndex DO
      IF Table[i]=NIL THEN GOTO found
      REPEAT
        found => new ← @Table[i];
        FINISHED => ERROR
      ENDLOOP;
    new↑ ← nextR;
    -- initialize the array storage
    new↑↑.p.backp ← new;
    new↑↑.p.length ← n;
    nextR←nextR+(n+ovh);
    END;

  Compact: PROCEDURE =  (omitted)

  FreeArray: PUBLIC PROCEDURE [dead: ArrayPtr] =
    BEGIN IF dead↑=NIL THEN ERROR; -- array already free
    dead↑↑.p.backp ← NIL;
    dead↑ ← NIL;
    END;

  -- Initialization
  i: TableIndex;
  FOR i IN TableIndex DO Table[i] ← NIL ENDLOOP;
  END.
```

Figure 2. Implementation of a compacting storage allocator

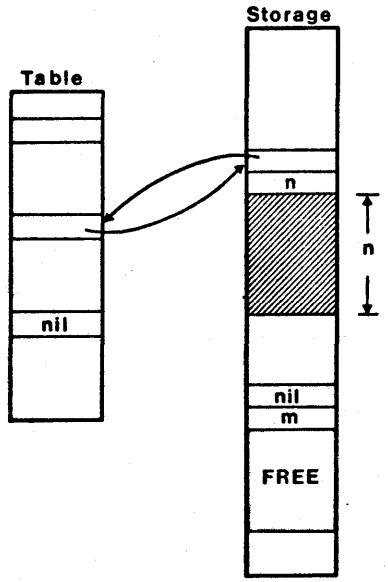


Figure 3. ArrayStore's data structure

Despite all these breaches we are still getting a good deal of checking. The checker would point out (or correct) any address/contents confusions we had, manifested by the omission of ↑'s or their unnecessary appearance. We can be sure that integers and *PR*'s are not being mixed up. In the (unlikely) event that we wrote something like

new↑*.p.length* ← *new*↑*.a*[*k*]

we would be warned because the value on the left is an integer and the value on the right is a *Thing*. Notice that none of this checking would occur if *Thing* were replaced by UNSPECIFIED. Thus even though the type system is not airtight we are better off than we would be in a completely unchecked language (unless, perhaps, we get a false sense of security).

Now let us consider how this module is to be used by a client who wants to manipulate two different kinds of arrays: arrays of integers and arrays of strings. At first it looks as if his code is going to have a very high density of RECAST's. For example, to create an array and store an integer in it he will have to say

IA: ArrayPtr = *AllocArray*[100];
IA↑↑*.a*[2] ← RECAST[6]

because the type of *IA*↑↑*.a*[2] is *Thing*, which doesn't match anything. Writing a loophole every time is intolerable, so we are tempted to replace *Thing* by UNSPECIFIED, thereby losing a certain amount of type checking elsewhere.

There are much nicer ways out of this problem. Rather than passing every array element through a loophole, one can pass the procedures *AllocArray* and *FreeArray* through loopholes (once, during initialization). The module *ArrayClient* (Cf. Figure 4) shows how this is done. Not only does this save our having to make *Thing* UNSPECIFIED, it allows us to use the type checker to insure that integer arrays contain only integers, and that string arrays contain only strings. More precisely, the type checker guarantees that every store into *IA* stores an integer. We must depend upon the correctness of the code in *ArrayStore*, particularly the compactor, to make sure that things stay well-formed.

```
    DIRECTORY ArrayStoreDefs: FROM "ArrayStoreDefs";
    DEFINITIONS FROM ArrayStoreDefs;

ArrayClient: PROGRAM =
  BEGIN
  -- Integer array primitives
  IntArray: TYPE = POINTER TO POINTER TO
      RECORD[p: Prefix, a: ARRAY [ 0..0 ] OF INTEGER];
  AllocIntArray: PROCEDURE [INTEGER] RETURNS [IntArray]
      = RECAST[AllocArray];
  FreeIntArray: PROCEDURE [IntArray]
      = RECAST[FreeArray];

  -- String array primitives
  StrArray: TYPE = POINTER TO POINTER TO
      RECORD[p: Prefix, a: ARRAY [0..0] OF STRING];
  AllocStrArray: PROCEDURE [INTEGER] RETURNS [StrArray]
      = RECAST[AllocArray];
  FreeStrArray: PROCEDURE [StrArray]
      = RECAST[FreeArray];

  Gedanken: PROCEDURE =
    -- This procedure's only role in life is to fail to
    compile if ArrayStore does't have the right sort of
    procedures.
    BEGIN
    uAllocArray:
      PROCEDURE [INTEGER] RETURNS [UNSPECIFIED]
        = AllocArray;
    uFreeArray: PROCEDURE [UNSPECIFIED] = FreeArray;
    END;

  -- no type breaches below here

  IA: IntArray = AllocIntArray[100];
  SA: StrArray = AllocStrArray[10];
  i: INTEGER;

  FOR i IN [0..IA↑.p.length) DO IA↑.a[i] ← i/3 ENDLOOP;

  SA↑.a[0] ← "zero"; SA↑.a[1] ← "one"; SA↑.a[2] ←
  "two"; SA↑.a[3] ← "surprise"; SA↑.a[4] ← "four";

  FreeIntArray[IA];
  FreeStrArray[SA];
  END.
```

Figure 4. The client of a compacting allocator

This scheme does not have any provisions for coping with problem (d), dangling reference errors. However, somewhat surprisingly, problem (e) -- saving a raw pointer -- cannot happen as long as the client does not commit any further breaches of the type system. The trick is in the way we declared *IntArray* -- all in one mouthful. That makes it impossible for anyone to declare a variable to hold a raw pointer. This is because (as mentioned before) every occurrence of the type constructor RECORD generates a new type, distinct from all other types. Therefore, even if we should declare

```
rawPointer: POINTER TO RECORD [p: Prefix, a: ARRAY[0..0] OF INTEGER ];
```

we could not perform the assignment *rawpointer* ← *IA*↑ because *IA*↑ has a different type, even though it looks the same. If one cannot declare the type of *IA*↑, it is rather difficult to hang onto it for very long. In fact, the compiler has been carefully designed to ensure that no type-checked program can hold such a pointer across a procedure call.

Passing procedure values through loopholes is a rather frightening thing to do. What if, by some mischance, *AllocArray* doesn't have the number of parameters ascribed to it by the client? Since we have waved off the type checker to do the assignment of *AllocArray* to *AllocIntArray* and *AllocStrArray*, it would not complain, and some hard-to-diagnose disaster would occur at run-time. To compensate for this we introduce the curious procedure *Gedanken*, whose only purpose is to fail to compile if the number or size of *AllocArray*'s parameters change. The skeleton type system, discussed earlier in this section, would obviate the need for this foolishness.

We would like to emphasize that, although our examples focus on controlled breaches of the type system, many real Mesa programs do not violate the type system at all. We also expect the density of breaches to decrease as the descriptive powers of the type system increase.

5. Variant records

Mesa, like PASCAL, has variant records. The descriptive aspects of the two languages' notion of variant records are very similar. Mesa, however, also requires strict type checking for accessing the components of variant records. To illustrate the Mesa variant record facility, consider the following example of the declaration for an I/O stream:

```
StreamHandle: TYPE = POINTER TO Stream;
StreamType: TYPE = {disk, display, keyboard};
Stream: TYPE = RECORD [
    Get: PROCEDURE[StreamHandle] RETURNS[Item],
    Put: PROCEDURE[StreamHandle, Item],
    body: SELECT type: StreamType FROM
        disk => [
            file: FilePointer,
            position: Position,
            SetPosition: PROCEDURE[POINTER TO disk Stream, Position],
            buffer: SELECT size: * FROM
                short => [b: ShortArray],
                long  => [b: LongArray],
                ENDCASE ],
        display => [
            first: DisplayControlBlock,
            last: DisplayControlBlock,
            position: ScreenPosition,
            nLines: [0..100]],
        keyboard => NULL,
        ENDCASE];
```

The record type has three main variants: *disk*, *display*, and *keyboard*. Furthermore, the disk variant has two variants of its own: *short* and *long*. Note that the field names used in variant subparts need not be unique. The asterisk used in declaring the subvariant of *disk* is a shorthand mechanism for generating an enumerated type for tagging variant subparts.

The declaration of a variant record specifies a type, as usual; it is the type of the whole record. The declaration itself defines some other types: one for each variant in the record. In the above example, the total number of type variations is six, and they are used in the following declarations:

```
r: Stream;
rDisk: disk Stream;
rDisplay: display Stream;
rKeyb: keyboard Stream;
rShort: short disk Stream;
rLong: long disk Stream;
```

The last five types are called *bound variant types.* The rightmost name must be the type identifier for a variant record. The other names are adjectives modifying the type identified to their right. Thus, *disk* modifies the type *Stream* and identifies a new type. Further, *short* modifies the type *disk Stream* and identifies still another type. Names must occur in order and may not be skipped. (For instance, *short Stream* would be incorrect since *short* does not identify a *Stream* variant.)

When a record is a bound variant, the components of its variant part may be accessed without a preliminary test. For example, the following assignments are legal:

```
rDisplay.last ← rDisplay.first;
rDisk.position ← rShort.position;
```

If a record is not a bound variant (e.g., *r* in the previous section), the program needs a way to decide which variant it is before accessing variant components. More importantly, the testing of the variant must be done in a formal way so that the type checker can verify that the programmer is not making unwarranted assumptions about which variant is in hand. For this purpose, Mesa uses a *discrimination* statement which resembles the declaration of the variant part. However, the arms in a discriminating SELECT contain statements; and, within a given arm, the discriminated record value is viewed as a bound variant. Therefore, within that arm, its variant components may be accessed using normal qualification. The following example discriminates on *r:*

```
WITH streamRec: r SELECT FROM
    display =>
        BEGIN streamRec.first ← streamRec.last; streamRec.position ← 73;
        streamRec.nLines ← 4;
        END;
    disk =>
        WITH diskRec: streamRec SELECT FROM
            short => diskRec.b[0] ← 10;
            long => diskRec.b[0] ← 100;
            ENDCASE;
    ENDCASE => streamrec.put ← streamrec.newput;
```

The expression in the WITH clause must represent either a variant record (e.g. *r*) or a pointer to a variant record. The identifier preceding the colon in the WITH clause is a synonym for the record. Within each selection, the type of the identifier is the selected bound variant type, and fields specific to the particular variant can be mentioned.

In addition to the descriptive advantages of bound variant types, the Mesa compiler also exploits the more precise declaration of a particular variant to allocate the minimal amount of storage for variables declared to be of a bound variant type. For example, the storage for *r* above must be sufficient to contain any one of the five possible variants. The storage for *rKeyb*, on the other hand, need only be sufficient for storing a *keyboard Stream.*

The mutable variant record problem

The names *streamRec* and *diskRec* in the example above are really synonyms in the sense that they name the same storage as *r;* no copying is done by the discrimination operation. This decision opens a loophole in the type system. Given the declaration

```
Splodge: TYPE = RECORD [
    refcount: INTEGER;
    vp: SELECT t: * FROM
        blue => [x: ARRAY[0..1000) OF CHARACTER],
        red => [item: INTEGER, left, right: POINTER TO Splodge],
        green => [item: INTEGER, next: POINTER TO green Splodge],
        ENDCASE];
```

One can write the code

```
t: Splodge;
P: PROCEDURE = BEGIN t ← Splodge[0, green[10, NIL]] END;
...
WITH s: t SELECT FROM
    red => BEGIN ... P[] .... s.left ← s.right  END;
```

The procedure *P* overwrites *t*, and therefore *s*, with a *green Splodge.* The subsequent references to *s.left* and *s.right* are invalid and will cause great mischief.

Closing this breach is simple enough: we could have simply followed ALGOL 68 and combined the discrimination with a copying operation that places the entire *Splodge* in a new location (*s*) which is fixed to be *red.* We chose not to do so for three reasons:

1. Making copies can be expensive.
2. Making a copy destroys useful sharing relations.
3. This loophole has yet to cause a problem.

Consider the following procedure, which is representative of those found throughout the Mesa compiler's symbol table processor.

```
Add5: PROCEDURE[ x: POINTER TO Splodge ] =
    BEGIN  y: POINTER TO green Splodge;
    IF x=NIL THEN RETURN;
    WITH s: x↑ SELECT FROM
        blue => RETURN;
        red => BEGIN s.item ← s.item+5; Add5[s.left]; Add5[s.right] END;
        green =>
            BEGIN y ← @s;   -- means  y ← x
            UNTIL y = NIL DO
                y↑.item ← y↑.item + 5;  y ← y↑.next;
                ENDLOOP;
            END
        ENDCASE;
    END
```

As it stands, this procedure runs through a *Splodge* adding 5 to all the integers in it. Suppose we chose to copy while discriminating: i.e., suppose $x\uparrow$ were copied into some new storage named *s*. In the *blue* arm a lot of space and time would be wasted copying a 1000 character array into *s*, even though it was never used. In the *red* arm the assignment to *s*'s *item* field is useless since it doesn't affect the original structure.

The *green* arm illustrates the usefulness of declaring bound variant types like *green Splodge* explicitly. If we had to declare *y* and the *next* field of a *green Splodge* to be simply *Splodges*, even though we knew they were always *green*, the loop in that arm would have to be rewritten to contain a useless discrimination.

To achieve the effect we desire under a copy-while-discriminating regime we would have to redesign our data structure to include another level of pointers:

```
Splodge: TYPE = RECORD [
    refcount: INTEGER;
    vp: SELECT t: * FROM
        blue => [POINTER TO BlueSplodge],
        red => [POINTER TO RedSplodge],
        green => [POINTER TO GreenSplodge],
        ENDCASE];
BlueSplodge: TYPE = RECORD[x: ARRAY[0..1000) OF CHARACTER];
RedSplodge: TYPE = RECORD[item: INTEGER, left, right: POINTER TO Splodge];
GreenSplodge: TYPE = RECORD[item: INTEGER, next: POINTER TO GreenSplodge];
```

Now we don't mind copying because it doesn't consume much time or space, and it doesn't destroy the sharing relations. Unfortunately, we must pay for the storage occupied by the extra pointers, and this might be intolerable if we have a large collection of *Splodges*.

How have we lived with this loophole so far without getting burnt? It seems that we hardly ever change the variant of a record once it has been initialized. Therefore the possible confusions never occur because the variant never changes after being discriminated. In light of this observation, our suggestion for getting rid of the breach is simply to invent an attribute IMMUTABLE whose attachment to a variant record declaration guarantees that changing the variant is impossible after initialization. This means that special syntax must be invented for the initialization step; but that is all to the good since it provides an opportunity for a storage allocator to allocate precisely the right amount of space.

6. Conclusions

In this paper, we have discussed our experiences with program modularization and strict type checking. It is hard to resist drawing parallels between the disciplines introduced by these features on the one hand and those introduced by programming without *goto*'s on the other. In view of the great *goto* debates of recent memory, we would like to summarize our experiences with the following observations and cautions:

1. The benefits from these linguistic mechanisms, large though they might be, do not come automatically. A programmer must learn to use them effectively. We are just beginning to learn how to do so.

2. Just as the absence of *goto*'s does not always make a program better, the absence of type errors does not make it better if their absence is purchased by sacrificing clarity, efficiency, or type articulation.

3. Most good programmers use many of the techniques implied by these disciplines, often subconsciously, and can do so in any reasonable language. Language design can help by making the discipline more convenient and systematic, and by catching blunders or other unintended violations of conventions. Acquiring a particular programming style seems to depend on having a language that supports or requires it; once assimilated, however, that style can be applied in many other languages.

References

1. Dahl, O.-J., Myhrhaug, B., and Nygaard, K., The SIMULA 67 common base language. Publ. No. S-2, Norwegian Computing Centre, Oslo, May 1968

2. Dennis, J.B. and Van Horn, E., Programming semantics for multiprogrammed computations. *Comm. ACM 9*, 3 (Mar. 1966), 143-155.

3. Geschke, C. and Mitchell, J., On the problem of uniform references to data structures. *IEEE Trans, SE-1*, 2 (June 1975), 207-219.

4. Habermann, A.N. Critical comments on the programming language PASCAL. *Acta Informatica 3* (1973), 47-57.

5. Knuth, D., *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.*, Addison-Wesley, Reading, Mass.

6. Koster, C.H.A., On infinite modes. ALGOL *Bulletin*, AB 30.3.3, (Feb. 1969).

7. Lampson, B., Mitchell, J., and Satterthwaite, E., On the transfer of control between contexts. in *Lecture Notes in Computer Science 19*, Goos and Hartmanis, ed., Springer Verlag, New York. (1974), 181-203.

8. Mitchell, J. and Wegbreit, B., Schemes: a high level data structuring concept. to appear in *Current Trends in Programming Methodologies*, R. Yeh, ed., Prentice-Hall, Englewood Cliffs, N.J.

9. Morris, J., Protection in programming languages. *Comm. ACM 16*, 1 (Jan. 1973), 15-21.

10. Parnas, D., A technique for software module specification. *Comm. ACM 15*, 5 (May 1972), 330-336.

11. Stoy, J.E. and Strachey, C., OS6 -- an experimental operating system for a small computer. part 2: input/output and filing system. *The Computer Journal 15*, 3 (Aug. 1972), 195-203.

12. van Wijngaarden, A. (Ed.), A report on the algorithmic language ALGOL 68, *Numerische Mathematik 14*, 2 (1969), 79-218.

13. Wegbreit, B., The treatment of data types in EL1. *Comm. ACM 17*, 5 (May 1974), 251-264.

14. Wirth, N. The programming language PASCAL. *Acta Informatica 1* (1971), 35-63.