Mesa: A Designer's User Perspective

James G. Mitchell

Xerox Palo Alto Research Center

Palo Alto, CA 94304

Keywords and Phrases: Data types, language evaluation, Mesa, Pascal, records, exception handling, systems programming, reliability, language design, programming tools

Abstract: Mesa is a fully typed language like Pascal, with some extensions. We here discuss three features of the language as originally conceived by its designers and describe how they have worked out in practice. The features are full data typing, facilities for constructing record values, and signals, an exception-handling mechanism.

1. Introduction

Mesa is the language component of a programming environment for developing and maintaining systems programs. Besides a compiler for the language, the environment includes tools for linking groups of modules together into programs, for loading programs, for debugging interactively, and for providing run-time support for programs. All of this environment is implemented in Mesa itself.

The language has been in active use for two years, and more than 100,000 lines of Mesa code have been written. They comprise interactive graphics programs, network communications software, compilers, debugging and performance-monitoring aids, and operating systems components (low-level device drivers, file systems, virtual memory support, and so on).

I intend to discuss here three of the language's features (data types in general, record types in particular, and an exception handling mechanism) as originally envisioned by one of Mesa's designers and how well they have worked out for users. Since I am now primarily a user and not a designer, I hope that this will remove some of the bias normally associated with "parents" of programming languages. Two previous papers [2][3] serve as good companions for this one and give more detail on Mesa's data type facilities and on the structure of modules.

Much of the information on which this report is based has come from conversations with other Mesa users. Much of it has also come from *history logs* that we are keeping at the end of each program to record changes to the program and their causes. These logs are valuable memory aids for reviewing a program's history, for seeing what kinds of errors have occurred most frequently, and for determining what kinds of errors were prevented by various language features. A *prevented* error is one that is caught by the compiler and prompts me to think what

would have happened had it lived on to become that most vile of creatures, a run-time bug. I recommend the keeping of such logs, no matter what language you program in; they are an excellent feedback mechanism for self-improvement as a programmer.

2. Using a Typed Language

Mesa is a fully typed language, much in the style of Pascal [6], with enhancements in the areas of record and array types (which will be discussed in section 3 below). Type checking has provided tremendous benefits for building large systems, especially those implemented by a group of people. The benefits derive from the fact that errors in using procedures or in accessing shared data structures are caught at compile-time rather than becoming run-time bugs.

For example, suppose one programmer implements a procedure, *SetItem*, that takes a pointer to some data item and stores a value (supplied as a second parameter) into it. If another programmer calls that procedure and mistakenly passes the item itself instead of a pointer to it, the compiler will complain because of the type mismatch between a value of type "POINTER TO *Item*" and one of type "*Item*". In an untyped language this error could propagate to run-time, in which case the called procedure would use the current value in the item as a pointer and proceed to smash the memory location that it designates.

Memory-smashing bugs such as these give programmers nightmares because they often do not show up until many instruction executions after the actual error. When they do manifest themselves, the symptoms seem calculated to lead us down various primrose paths completely unrelated to the actual problem. They are costly to repair and often lie latent long after the culprit program is deemed to be debugged, as if waiting for the optimum time to invoke Murphy's Law and make their presence known.

Type checking can also detect bugs resulting from supplying the arguments to a procedure in an incorrect order. For example, a memory-smashing bug similar to the above example could occur if the procedure *SetItem* were called with the order of the value and pointer parameters reversed, even though their types were correct. In a strongly-typed language, this error is detected at compile-time.

If the types of some of the parameters to a procedure are the same, however, even type checking may be insufficient to detect such errors of ordering. For example, a procedure, *CopyItem*, that takes pointers to two *Item*s as parameters and copies the value of one of the items into the other is a common culprit - some such procedures store the value for the first parameter into the second, while others do just the opposite. This kind of error won't result in memory smashing but it is a run-time error and will probably require some detective work to find. In Mesa, the actual parameters to a procedure may be associated with the formals by *name* instead of by the positions of the actuals in the parameter list. This fixes the problem because the caller need only use the formals' names, *to* and *from* for instance, and may ignore their order (Aside: this is a good feature even if a language is not typed, as in the IBM System/360-370 Assembly Macro facility[5]). This *keyword naming* feature is described in more detail in section 3.

2.1. Unanticipated benefits of type checking

When we designed Mesa, we expected benefits from type checking of the sort that I have just described. What we didn't anticipate was its utility in program modification, and we have been pleasantly surprised at the relative ease of changing interfaces in systems of Mesa modules without introducing bugs in already working programs. For example, we have changed some of the interfaces between the modules in a system, then changed the system as necessary to accommodate the altered interfaces, and recompiled the modules affected by the changes. This has, with very high reliability, produced a system that still works - spurious bugs are not introduced. Moreover, it has enabled programmers to more reliably change each other's programs.

Both these effects result from the compiler's *insistence* on type-correct programs. If you forget to modify a call on some procedure whose parameter requirements have changed, the compiler will let you know; if you don't change the call correctly, the compiler will let you know; if you try to store the result of a function into the wrong type of variable, the compiler will let you know; if you forget that a procedure is a function and forget to store its result, the compiler will let you know.

It is only a matter of time before fully typed languages such as Mesa and Pascal come into more common use on minicomputer and microcomputer systems. Our experience suggests that implementing Pascal's surface syntax without providing full type checking removes a large part of its value for writing software that works. If I had to choose between structured control constructs (such as WHILE, IF-THEN-ELSE, etc.) and having my programs type checked, I would vote for the type checking. It is relatively easy to use unconstrained GOTOs in the "approved, structured" ways; it is much more tiresome to check for all the type errors that one can make in a language such as Mesa or Pascal with their rich data-structuring facilities.

Most of the memory-smashing bugs that have occurred in Mesa programs have been traced to two sources: uninitialized variables and bypassing the type machinery. Uninitialized pointers and index variables account for virtually all of the first kind of problems (in [1], similar results are shown). Automatically initializing all pointer variables (including components of records) to NIL and trapping when NIL is used to reference some datum would take care of the first of these. Run-time (or better yet, compile-time) bounds checks on stores into subrange variables would catch those of the second kind.

The issues surrounding users' need to bypass the Mesa type machinery are outside the scope of this report. They are discussed in detail in [3].

3. Records and Arrays as Scalars

Records and arrays in Mesa are basically the same as in Pascal, but they can be manipulated more readily in Mesa. Since the facilities available for records and arrays are similar, we will only discuss those for records here. The major differences between Mesa and Pascal in treating records are

(1) Records and arrays are treated like first-class scalars in Mesa; in particular, one can always compare two records (of the same type, of course) for equality, assign one to the other, pass them as parameters to procedures, or return them as results from functions.

(2) One can *construct* a record value as a scalar by writing a list of values for its components. The converse capability allows one to *extract* multiple fields from a record value with a single assignment statement.

(3) The parameters for procedures are treated as records, as are the return values of functions. Thus, any facilities provided to ease the construction of record values (e.g., those developed for (1)) can be used for writing parameter records when calling procedures. More importantly, however, the programmer need only learn one set of syntax and semantics for dealing with records and for passing parameters (and receiving results) from procedures.

We will provide more detail on these features with some examples.

3.1. Record constructors and extractors

A Mesa constructor is a list of values enclosed in brackets. The values may be associated with fields of the record type in one of two ways: by position, or by keyword. A positional constructor looks just like a parameter list for a procedure call in most languages; a keyword constructor explicitly specifies which field each component value is to be assigned to by naming them. Here is an example of a record type *Time*, two *Time* variables *start* and *finish*, and two constructors, a positional one assigned to *start*, and a keyword one assigned to *finish*:

```
Time: TYPE = RECORD
 [
 hour: [1..12],
 minute, second: [0..60),                    -- two fields with same type
 meridian: {am, pm}
 ];                                          -- end of Time

start, finish: Time;                             -- declare two variables
start ← [10, 30, 00, am];                    -- value for 10:30:00 a.m.
finish ← [hour: 10, minute: 30, second: 00, meridian: am];   -- also means 10:30:00 a.m.
```

Section 2 described how keyword parameter lists help in avoiding errors in providing actual parameters to procedures. That holds for all record constructors. In particular, if the order of *minute* and *second* in a *Time* were reversed, the constructor assigned to start would still be type-correct (since those two fields have the same type), but it would describe the time 10:00:30 a.m., which is not what is intended. However, the keyword constructor assigned to *finish* would still be the correct value for the time 10:30:00 a.m. because it is order-independent.

An extractor allows one to assign the fields of a record value to separate variables with a single assignment statement. For example, we could store the components of the variable *start* into four variables, *hr, min, sec,* and *ampm* with either a positional or a keyword extractor as follows:

```
[hr, min, sec, ampm] ← start;      --positional extractor
[hour: hr, minute: min, second: sec, meridian: ampm] ← start;
```

Here again, the keyword version has methodological advantages over the positional. It avoids ordering errors, and it does not require the reader to refer to the type definition for *start*'s type (which may not be nearby) to know the order of the fields.

Positional notation is useful when a constructor or extractor has only one or two components and the types are different. We did not enforce any such discipline in Mesa, but many experienced Mesa programmers are now independently adopting such a standard after being caught by positional errors.

We erred, too, in not providing a form of keyword constructor for array values. It was in the original language design, but was deemed less useful than record constructors (partly because of the predominance of record values in the guise of procedure parameter lists). I have had run-time bugs that were a result of writing array constructors with values out of order. To avoid such problems, I now tend not to initialize arrays with constructors, but I would rather have keyword array constructors with the compiler checking them for me (because it is more diligent at watching for errors than I am).

3.2. Procedure parameters and results

The parameters to a procedure and the results from a function are just records in Mesa. For convenience they do not have to be declared separately from the procedure declaration, but they are bona fide record declarations nevertheless. For example, one can declare a procedure *LookUp* to search a table (of type *Table* naturally) for a *key* and return a value, *val,* stored with that *key* in the table. We choose to insert the *key* if it is not already present in the table and return a second result, *alreadyIn,* along with *val* to indicate whether it was entered anew or was already in the table. *LookUp*'s declaration is

```
LookUp: PROCEDURE[t: Table, key: STRING]
                RETURNS[val: INTEGER, alreadyIn: BOOLEAN] =
BEGIN
...                                  -- the body of the procedure
END;
```

Most systems programming languages do not allow one to return such multiple values, and generally for good reason: it is not clear what to do with such a multi-component value when it *is* returned. Mesa's extractors and qualification both help with this. A program that needs to store both the result values for further processing can use an extractor; for example,

[*alreadyIn*: *isNew*, *val*: *keyVal*] ← *LookUp*[*directory*, "J. G. Brannan"];

A program that does not care whether the *key* is being entered for the first time can use *qualification* to access the one value required. Alternatively, it can use an extractor, omitting the unwanted results. For example, the following two statements each use only the *val* result from *LookUp*:

```
[val: keyVal] ← LookUp[directory, "J. G. Brannan"];-- omit storing alreadyIn
IF LookUp[directory, "J. G. Brannan"].val < 0 THEN WriteString["Error in directory"];
```

These facilities have worked out well in Mesa and are heavily used. There is often a price for success, however: Because of the ease with which one can pass arguments that are themselves entire records, the average length of parameter lists has increased in Mesa over the average in an earlier version that did not have these features. This has necessitated some redesign of the parameter-passing mechanisms to reduce the cost of passing long parameter lists.

The only moral that I am able to draw from this is that programmers tend to be influenced by the number of characters that they must write in order to accomplish some desired action. Thus, language designers must be careful if they make an inefficient feature easy to write. Its ease of use will increase its frequency of use and may require extra effort from the compiler to make the implementation of the feature sufficiently efficient to stand up to its popularity.

4. Exception handling: Signals

In even the best designed software systems, errors and exceptional conditions arise. To handle such cases, Mesa has *signals*.

Usually, some exceptional conditions are part of the specifications of a program, but most are "never supposed to happen". An error in a disk operation, running out of space in memory, or incorrect input from a user of the program are exceptions that should be anticipated in a program and handled by the program itself. However, an incorrect parameter to a procedure (we can be confident that its type is correct, but not necessarily its value), or attempting to read a file that has not yet been opened are generally not planned for and usually indicate errors in the program. Signals are used for both these purposes.

If a procedure detects that something is amiss, it can generate a SIGNAL or an ERROR (which we will lump together under the term signals). For example, assume that a procedure *ReadChar* contains the following statement to generate the ERROR *BadCharacter* and passes along the *input* character that looked fishy:

```
ERROR BadCharacter[input];
```

The signal is then sent to the nearest outstanding procedure (call it *P*) in the call chain that has provided a *handler* for that signal. Basically, a handler is a procedure local to that outstanding procedure, *P*. The handler is called and passed parameters given when the signal was generated.

A handler can be attached to a particular call of a procedure (it can also be placed on a whole block, but that is a convenience and not critical here). For example, *P* might have called *ReadChar* in the following way, indicating that it wanted to transfer control to the place named *Retry* if *ReadChar* generated the signal *BadCharacter*:

ReadChar[*inputString* ! *BadCharacter* => GOTO *Retry*];

A handler has all the capabilities of any good procedure plus one extra: it can initiate a non-local GOTO into the body of the outstanding procedure in order to reset control back to that procedure. This is not done as an unconditional jump; rather, each procedure instance from the one that generated the signal up to *P* will have a chance to handle a special signal, UNWIND. UNWIND serves as notice to a procedure instance that it is about to irrevocably lose control and provides it with a chance to clean up any data structures external to itself, to close files, etc. If a procedure never has any such side effects, of course, it need not handle UNWIND at all. When all the intervening procedure instances have been disposed of, the non-local GOTO completes and *P* gains control at the place specified by the GOTO.

A handler can react to a SIGNAL in one way that is different from how it can react to an ERROR: it can tell the procedure that generated to RESUME processing (and can actually pass a value to it as if the SIGNAL were a procedure that the generator had called). This is useful if the handler can somehow fix the problem that gave rise to the signal or if the signal is being used more to notify a handler than to drastically alter the flow of control.

These facilities have been adequate for our needs, even though they span but a small part of the exception handling facilities envisioned by Goodenough [4]. However, they are not right for handling exceptions arising in a multi-process environment because Mesa contains no language facilities for directing signals from one process to another. The search for a handler for a signal follows the procedure call hierarchy only.

What if no procedure is interested in some generated signal? Well, Mesa provides a backstop for all such signals and calls the interactive debugger, which then reports the signal to the user. The important point about this is that all the state that existed when the signal was generated is still around and can be perused by the programmer to help in finding the source of the difficulty. Since it is so easy to specify that a signal be generated, programs use this facility to indicate any untoward program states, "funny" parameters, etc. This has helped greatly with debugging and with tracking down bugs that occur even after a program has been in general use for some time.

Almost all the signals ever generated have been used for this latter, debugging, purpose. While this is extremely useful, I had expected that more programs would use them in the way that was outlined for the *ReadChar* example. The only explanation that I have for this lack of enthusiasm for signals is that most programmers have no experience with them at all since they are a new addition to most systems programming languages. Record constructors, while also new, are really just extensions on an already well understood base, structured data objects, and have certainly been more widely accepted and incorporated into programs.

Thus, it seems clear that the signal mechanisms in Mesa need work, although it is not clear exactly how they should be modified. What is clear is that having *any* standardized exception-handling mechanism is vastly superior to having none at all.

5. Conclusions

We have reported on three aspects of the Mesa language and how they have worked out in practice. The first, full type checking of the data in programs has changed certain classes of run-time errors to compile-time errors, with a concomitant decrease in the cost (measured in programmer time) of producing programs. We were surprised that it had a large effect on the ability to modify programs without introducing new bugs in the process.

The second aspect, expanded record facilities, as well as increasing the expressive power of the language has also had an effect in changing positional errors in procedure-call parameter lists into compile-time rather than run-time errors.

Lastly, the Mesa exception handling mechanisms, signals, have proved useful, especially for debugging and for flagging those conditions that should "never" happen, but do. Their use for dealing with exceptions within a program completely automatically has been less than expected, although they appear adequate for this purpose.

6. Acknowledgements

The principal designers of Mesa, in addition to the author, have been Charles Geschke, Butler Lampson, and Edwin Satterthwaite. Major portions of the Mesa run-time support software were programmed by Richard Johnsson and John Wick of the System Development division of Xerox. Lastly, my experiences as a user have benefitted greatly from working with Jay Israel, James Morris, and Howard Sturgis.

7. References

[1] Endres, A. B., An Analysis of Errors and Their Causes in System Programs, *IEEE Trans. Software Eng. Se-1*, 2 (June 1975), 140-149.

[2] Geschke, C. and Mitchell, J., On the Problem of Uniform References to Data Structures, *IEEE Trans. Software Eng. Se-1*, 2 (June 1975), 207-219.

[3] Geschke, C., Morris, J. H., and Satterthwaite, E., Early Experience with Mesa, *Comm. ACM 20*, 8 (August 1977), 540-552.

[4] Goodenough, J. B., Exception Handling: Issues and a Proposed Notation, *Comm. ACM 18*, 12 (December 1975), 683-696.

[5] IBM System/360 Operating System, Assembler Language, *Form C28-6514-4*, Poughkeepsie, NY, (1964).

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