Designing Loupe: A Modula-2$^+$ Debugger

John DeTreville
Systems Research Center
Digital Equipment Corporation

March 18, 1986

1 Introduction

Loupe is the symbolic debugger for Modula-2$^+$ (the dialect of Modula-2 developed and used at DEC's Systems Research Center). Loupe provides most of the features one would expect from a modern high-level language debugger–mostly in the ways one would expect–but with some number of interesting capabilities. Loupe understands the Modula-2$^+$ type system; you (the user) can use Loupe to inspect and modify the data values in your program. Loupe also treats types and modules as values, and lets you inspect them. You can type the names of variables and get their values; you can also evaluate arbitrary Modula-2$^+$ expressions. Loupe provides a keyword-oriented command language whose design was iterated until it became easy to use. Loupe itself is programmable, as well as providing mechanisms for controlling the execution of the program being debugged. The design of Loupe evolved over time, guided by its users' reactions.

2 Displaying Values

Loupe understands the Modula-2$^+$ type system; you can use Loupe to inspect and modify the data values in your program.

The output syntax for a value is the same as the input syntax. Some values can be input in multiple ways (`144C` = `'d'` = `"d"` = `'\144'` = `"\144"`); these are canonicalized on output (`'d'`, but `"d"` for the one-character string as distinguished from the character constant, and `0C` for the null character, and so on).

Modula-2+ has no input syntax for values of some types. For example, RECORD values have no input syntax, nor do ARRAY values (except for character arrays in their role as character strings), nor do POINTER or REF values, nor do System.Address values, and so on. Loupe provides no input syntax for such values (except for circumlocutions like LOOPHOLE(929e4H, System.Address)), but it manufactures an output syntax.

RECORD values print as a list of name-value pairs, as in <seconds = 509747164, microseconds = 210000>. Array values print as a list of values, as in <0, 10, -11, 1, 0, 0, 8, -10, 15, 3>. Angle-brackets are used to enclose these lists because angle brackets were otherwise unused.

Early in Loupe's development, RECORD and ARRAY values were displayed prefixed by their type name, if any; this was by analogy to SET values. Back then, Loupe would display the above RECORD value as Time.T<seconds = 509747164, microseconds = 210000>. Users complained that this was too verbose ("If I wanted to know its type, I'd have *asked* for its type"). When the analogy with SET values was explained, users suggested that maybe SET values shouldn't print with type names either. In the current compromise, Loupe prints SET values with a leading type name, but RECORD and ARRAY values without.

What about POINTER values and REF values? Loupe initially took the view that POINTER and REF values were just indirect names for the values they referenced, so a POINTER value might print as POINTER TO <seconds = 509747164, microseconds = 210000>. Circular structures were elided, as well as very long or deeply nested structures (under simple user control). An opposing view was that POINTER values and REF values are just addresses, and should print in hexadecimal; the above POINTER value might print as 08ce18H. Of course, the value of the referent and the address of the referent can be independently obtained by the Loupe user; the question was what Loupe's default output format should be. This became a religious issue, with neither side wanting the output syntax to detract from the true meaning of POINTER and REF values. The current compromise is to print POINTER and REF values as in POINTER TO <Time.T at <08ce18H>>, which greatly displeases only the "If I wanted to know its type, I'd have *asked* for its type" camp.

Loupe checks POINTER and REF values for consistency before printing them; the referent must be accessible in memory (as might not be the case for an uninitialized POINTER or REF), and the typecode stored with a REF's referent must match the REF's type. Loupe initially printed bad POINTER and REF values as hexadecimal numbers (equating a typefree format with a

typefree value) but users didn't get the joke; Loupe now prints such values as, say, `<the nonsensical TextCommon.CharPtr value 020202020H>`.

`REFANY` values print with the name of the actual `REF` type prefixed, as in `Wr.T: REF <WrV.T at <08ca74H>>`; this happened not to be controversial.

`System.Address` values print symbolically when possible. If the address is a text address within the target program, a reference into the program is printed, as in `<Scanner.NextChar, line 63 ("c := Rd.GetChar(in);") + 0cH>`. If it is a static data address, the name of the data object is printed, as in `<ThreadsPort.readCV[0].queue>`. Local data addresses (stack- or heap-based) are not printed symbolically, for reasons of efficiency. Fallback positions are to print addresses relative to a linker symbol (`<_SYSTEM_transfer + 01daH>`) or in hexadecimal (`<085618H>`).

Since `System.Address` values print symbolically, you can print an arbitrary number symbolically by casting it to a `System.Address`. Similarly, an arbitrary number can be printed in hexadecimal by casting it to a `System.Word`, or in decimal by casting it to an `INTEGER`. This was once the only way to choose output format, but users complained that one should be able to specify the format independent of the type. Another religious war ensued. Loupe now lets you specify the format independent of the type: formats are `decimal`, `octal`, `hexadecimal`, `address`, and `roman` (a joke on the proliferation of formats, but few users read the documentation in enough detail to notice it; Loupe prints 1986 in `roman` as `0mcmlxxxviR`, and accepts this input syntax as well).

Loupe currently provides a `hexdump` command, which prints the contents of a range of addresses in hexadecimal and ASCII. The initial Loupe policy position was that such a command was unnecessary since one could (and should) access memory through the program's type structure, but this is sometimes impractical; one common use of the `hexdump` command is to look for some pattern in an area of memory that has unknown or unexpected contents.

Loupe's current tty-style user interface places a premium on the use of screen space; the less it displays, the better. On the other hand, if you are forced into multiple steps to obtain some desired output, the screen will scroll faster than if the output were more simply available. There is generally a balance between brevity and verbosity, arrived at historically; a value that now displays as `POINTER TO <Time.T at <08ce18H>>` would once have displayed as `POINTER TO <the Time.T value at 08ce18H>`.

Another balance is between brevity and readability. Loupe currently provides a simple pretty-printer for its output; the `RECORD` value `<definition =`

```
<in = REF <ScopeDef.ScopeR at <08b244H>>, name = "CARDINAL">, tag
= TypeDef.SubRangeTag, subRange = <type = REF <TypeDef.TR at <08cb24H>>,
a = 0, b = 2147483647>>
```

might print

```
<definition =
   <in =
      REF
        <ScopeDef.ScopeR at <08b244H>>,
    name = "CARDINAL">,
 tag = TypeDef.SubRangeTag,
 subRange =
   <type =
      REF <TypeDef.TR at <08cb24H>>,
    a = 0, b = 2147483647>>
```

(on a narrow screen). Most users find the pretty-printed form worth wasting screen space for, and others can disable the pretty-printer.

3 Values

Loupe's idea of value extends Modula-2^+'s. For example, types are considered values; if you ask for the value of `System.Address`, you get `POINTER TO System.Word` (if you ask for the value of `System.Word`, you get `<WORD>`, which is Loupe's internal model of the `System.Word` type). Loupe provides a new built-in procedure `TYPEOF`; asking `TYPEOF(99)` gives `[99..99]` (this follows Modula-2^+'s rule for numeric constants). Asking `TYPEOF(System.Word)` gives `<INTERNAL>`, since it's an object internal to Loupe.

Loupe models values and variables as *(type, locative)* pairs. Locatives are also user-visible values. Asking `LOCOF(Writer.depth)` might give `<the 32 bits at <loc 05d740H>>`; asking `LOCOF(TRUE)` on a VAX gives `<the bits representation in <the bits 10000000>>` (Loupe's view of an immediate value). The locative can be as complex as

```
<the open array of 8-bit elements with
 count at
 <the 32 bits at
  <32 bits off
   <the local at <64 bits off <ap>> with
    fp = 0926f8H and ap = 09270cH>>>
 and contents at
```

```
<indirect thru
 <the 32 bits at
  <the local at <64 bits off <ap>> with
   fp = 0926f8H and ap = 09270cH>>>>
```

Locatives are interesting to look at (if the user doesn't want interesting, the user can use `System.Adr` instead of `LOCOF`). Locatives can also be used wherever `System.Address` values can be used, if they lie on a byte boundary (locatives provide bit-addressing internal to Loupe).

Modules and procedures are also values. The module `Token` might display as:

```
<<T =
  REF
    RECORD
      CASE tag: TokenDef.Tag OF
      | TokenDef.ValueTag: value: V.T;
        ELSE name: TextCommon.Text; END;
    END>,
 <_initflag =
  <the BOOLEAN variable at
   <loc 0695a0H>>>,
 <Prin =
  <the PROCEDURE(REFANY) at
   <loc 02ef08H> with
   <<x =
     <the REFANY argument at
      <32 bits off <ap>>>>,
    <token =
     <the Token.T variable at
      <-288 bits off <fp>>>>>>,
 <_init =
  <the PROCEDURE() at <loc 02f038H> with
   <>>>>
```

This mechanism lets you browse through the name space. (Here, `_init` is the compiler-generated "initialization procedure" corresponding to `Token`'s body; `_initflag` is used at runtime to avoid executing `_init` twice. Loupe does not attempt to hide these artifacts from the user.)

Loupe also treats `EXCEPTION`s as values. `System.Fail` would be displayed as `<an EXCEPTION(System.FailArg)>`.

Loupe does not consider stacks or stack frames to be values; this seemed too difficult. As a result, the user mechanisms for browsing and manipulating control structures are distinct from those for data structures.

4 Names

You can type the names of variables to Loupe and get their values. As a default, top-level names are looked up in the global scope. Thus, the name `Module` refers to the top-level module `Module`; the name `System` refers to the `System` module; the name `INTEGER` refers to the built-in type `INTEGER`; and so on.

If a procedure activation is selected, names defined in it are visible first, then the names in its enclosing scope (a module or a procedure), and so on up to the global naming scope. Since names in outer scopes may be hidden by names in inner scopes, Loupe predefines the (fairly unique) name `@` in the global scope to refer to the global scope itself; a fully qualified name can therefore be preceded with `@.` since `@` will probably be defined in no inner scope.

A qualified name, such as `System.MaxCard`, is interpreted by first looking up `System`, then interpreting `MaxCard` in that context. The left-hand side of the . operator can be a module or a `RECORD` value, as in Modula-2$^+$; Loupe also allows it to be a `PROCEDURE` value, in which case the stack is searched for an activation of that procedure and the right-hand side is interpreted in its context.

Loupe does not provide unqualified dynamic scoping. If you ask for `i`, and there is no `i` in the selected activation, Loupe does not search up the stack for any `i`. Although such automatic searching would seem friendly in some instances, it would be probably be too dangerous in general.

Loupe draws no distinction between names defined in an `IMPLEMENTATION MODULE` and names defined in a `DEFINITION MODULE`; information-hiding seems undesirable in a debugger.

When Loupe prints a value, it might choose to print it by name. Types can have names; asking `TYPEOF(3+4)` displays `INTEGER`. The official name of a type is the first name bound to that type; if a program defines type `A` to be `POINTER TO System.Word`, and type `B` to be `A`, and type `C` to be `B`, then asking Loupe for `C`'s value will display `A`. As a special case, asking Loupe for `A`'s value will display `POINTER TO System.Word`; Loupe never answers a question with the same question. Note that if Loupe displayed `C`'s

value as `POINTER TO System.Word`, the user would have no way of knowing that it was the same instance of `POINTER TO System.Word` as `A`; this is an important distinction given Modula-2$^+$'s type semantics.

Scopes (modules and procedures) always have names, and they are always printed by name except in response to a direct question.

Enumerated values also have names, and are printed by name. Enumerated values that are out of range (*e.g.*, a `BOOLEAN` whose `ORD` is 3) print in hexadecimal.

Otherwise, values are not printed by name; Loupe will never print `System.MaxCard` instead of 2147483647. An early version of Loupe would have, but the rules for equality for ordinary values make any such mechanism impossible or confusing.

Names printed by Loupe are fully qualified; a fully qualified name is always possible since scopes always have names. At one time, Loupe qualified names only enough for the selected context, but this seemed potentially confusing to users; a name printed out a few interactions ago may no longer be fully correct. Loupe's current rule is justified by the maxim that the output format should be an input format.

5 Expressions

Loupe contains a full Modula-2$^+$ expression evaluator, with a few extensions to increase its usefulness.

Loupe's expression syntax is more general than Modula-2$^+$'s. Modula-2$^+$ doesn't allow the user to say `Time.Now().seconds`, thereby simplifying compiler-writing while simultaneously enforcing programming rigor. These are not issues for Loupe, and the construct is useful interactively, so Loupe allows the construct.

Similarly, you can say `LOOPHOLE(x, ARRAY [0..7] OF BITS 4 FOR CARDINAL)`. Again, this is convenient in interactive use even though it may be undesirable in a program.

Loupe's expression syntax was derived from Modula-2$^+$'s by such small generalizations. Loupe uses a simple recursive-descent parser, whose flexibility has often been convenient, and whose simplistic error detection and correction pose no real problem.

Loupe's lexical scanner began as an exact copy of the compiler's (and its use in an interactive environment helped speed the discovery of several bugs). Changes to the scanner included extending the set of characters allowed in

identifiers (Loupe provides partial support for programs in languages other than Modula-2+, such as C and assembler, with different lexical rules for identifiers); additionally, any character is considered alphabetic if \-escaped.

Semantically, Loupe's expression evaluation also extends Modula-2+'s for the sake of interactive convenience. `REFANY` values, for example, can be dereferenced without being `NARROW`ed to the correct `REF` type; Loupe automatically performs the `NARROW`.

Similarly, an array can be subscripted by an appropriate subrange type, obtaining a subarray value; `"abcd"[[1..2]]` is `"bc"`. (Because of its syntax, users tend to remember this facility as "Use double brackets to get a subrange").

Loupe implements all Modula-2+ built-in procedures except for `DISPOSE`, `NEW`, `RAISE`, `System.NewProcess`, `System.Transfer`, and `System.UnixCall`; implementing these correctly would require additional interaction between Loupe and the runtime system. Many of Modula-2+'s built-in procedures have proved quite handy interactively; you can ask, for example, `System.Size(x)`.

Many new features of Loupe are provided by built-in procedures. `TYPEOF` and `LOCOF` have already been mentioned. The built-in procedure `AT(`*location*`,` *type*`)` returns the *type* value at *location*. (This order of arguments is by analogy with `LOOPHOLE`. Loupe also accepts `AT(`*type*`,` *location*`)`, by analogy with `VAL`. The author likes to be able to use either order without thinking, and usually does.)

`LINE(`*procedure*`,` *n*`)` returns the `System.Address` of line *n* in *procedure*, which is sometimes useful. `REGISTER(`*n*`)` returns the value of register *n* in the selected activation. `GETWORDAT(`*location*`,` *n*`)` fetches the *n*-bit word at *location* in a single operation, facilitating access to device registers; `SETWORDAT(`*location*`,` *n*`,` *value*`)` stores a new value in a single operation.

Some operators and built-in procedures provided by Loupe have looser type-checking than in Modula-2+; this was for ease of implementation in Loupe. For example, one can say `TRUE OR 5`, which evaluates to `TRUE`; evaluating `TRUE AND 5` gives a type error.

6 Command Syntax

If Loupe provided a full Modula-2+ interpreter, Loupe's command language would then be Modula-2+. On the other hand, providing a Modula-2+ interpreter, while quite useful in many ways, would also be quite an undertaking, and it is unclear how good a command language Modula-2+ would be.

Loupe's command language was initially based directly on expression evaluation, as in Lisp systems. Typing an expression caused it to be evaluated and the result, if any, printed. The expression syntax was slightly extended to include assignment; other operations were performed by special built-in procedures.

So, to print x, you would type simply

```
<1> x
```

(Loupe prints numbered command prompts in angle brackets) and Loupe might reply

```
4
```

To set x to 5, you would type simply

```
<2> x := 5
```

To set a breakpoint at line 63 of `Scanner.GetChar`, you would type simply

```
<3> BREAK(LINE(Scanner.GetChar, 63))
```

Users complained unanimously, and Loupe eventually went to a more conventional keyword syntax; the examples above became

```
<1> print x
<2> set x := 5
<3> break LINE(Scanner.GetChar, 63)
```

After more complaints, the last finally became

```
<3> break Scanner.GetChar 63
```

The `print` command prints the value of an expression; the `set` command performs an assignment. In the original expression-oriented command syntax, you could type an expression with no value, such as `INC(i)`, and no value would be printed. Since `print` would be a misnomer for such an operation, the `call` command was added; it takes an expression that returns no value (which, in Modula-2^+, must be a procedure application; an empty argument list is added if absent) and merely evaluates it. To enforce proper rigor, the `print` command signaled an error if the expression had no value, and `call` signaled an error if the expression did; this was later relaxed to make `print` and `call` identical except for their names and for `call`'s syntactic insistence on a procedure call.

The next change was to add abbreviations. Some commands, like `print`, are so heavily used that even such a short name is cumbersome. Other commands, like `previousframe` (formerly the built-in procedure `PREVIOUSFRAME`), are cumbersome to type even infrequently. A system of two-letter abbreviations was instituted: `print` can be abbreviated `pr`, `previousframe` can be abbreviated `pf`, and so on. All abbreviations are two letters long, for consistency. Some commands were not abbreviated; `set` has no reasonable two-character abbreviation, and although `break` could have been abbreviated `br`, the closely related `unbreak` could not have received the closely related `ubr`, and `ub` seemed too unpleasant. Similarly, neither the `frame` nor the `frames` command were abbreviated.

In the current version of Loupe, a space at the beginning of a command line is taken to mean `print`, `set` or `call`; this almost returns to the original syntax for these common, simple cases. Instead of typing

```
<4> print x
```

(Loupe prints numbered command prompts in angle brackets) or

```
<5> pr x
```

you can use a leading space and type simply

```
<6>  x
```

The initial space is very easy to type; it becomes a reflex. In fact, it becomes so much of a reflex that users often type an initial space before other commands; this is an error. In correcting for this behavior, they start leaving off the space in cases when they should have it; this is also an error. It would be nice if Loupe tried to second-guess such behavior and correct for it; one simple implementation of this idea, though, turned out to produce confusing diagnostics for syntax errors not involving extra or missing leading spaces.

Commands can extend over more than one line. Loupe uses the rule, borrowed from BCPL, that a new-line ends a command if the command could syntactically end at the new-line. So, you can say

```
<7> print 3+4
7
<8> print 3+
    4
7
```

In practice, this works quite well. Implementing this mechanism, and the leading-space mechanism, required close cooperation between Loupe's lexical scanner and its parser.

In all cases, Loupe treats the command syntax merely as an alternative to the expression syntax, and turns all commands into applications of built-in procedures. Commands and expressions therefore differ only syntactically. As an escape from expression syntax into command syntax, Loupe provides the `!(` and `)` brackets; enclosing a command in `!(` and `)` allows it to appear in an expression context. Similarly, the `value` command, which returns the value of its argument, allows an expression to appear in a command context.

Why is the ability to switch between syntaxes important? Commands can take arguments that can be expressions or can be other commands. For example, Loupe's `if` command should take its first argument, the predicate, in expression syntax rather than in command syntax; `if`'s consequent arguments would most naturally appear in command syntax, by analogy to Modula-2^+'s `IF` statement. For some arguments to some commands, though, it is less obvious whether they should by default be in expression syntax or in command syntax. Loupe's simplistic solution is that all arguments to all commands appear in expression syntax. This rule is easy to remember, although it means that you may have to type `!(` and `)` more often than otherwise. (This solution also has the amusing property that the `end` in `if ... then ... else ... end` and in similar constructs is optional since only single expressions appear as the arguments.) Operators take their arguments in expression syntax, except for the `;` operator, which takes its in command syntax.

Let's look at some individual commands. The `print` command has the form `print` *[expression [format] [, ...]]*. Each *expression* can be followed by a *format*: `decimal`, `octal`, `hexadecimal`, `address` or `roman`, or their three-character abbreviations `dec`, `oct`, `hex`, `adr` or `rom`. Since the *format* is a single identifier, no syntactic ambiguity results. One `print` command can print multiple *expressions*, separated by commas; this is handy for tabular output. A trailing comma suppresses the trailing new-line on output.

The syntax of the `hexdump` command is `hexdump` *[address] [, count]*; defaults are chosen for the *address* and the *count* if they are absent. An earlier syntax which did not include the comma was syntactically ambiguous; `hexdump ptr (newCount - oldCount) * 4` could be parsed incorrectly. If Loupe had used a parser mechanically derived from a formal grammar, this ambiguity could have been detected, but Loupe's syntax would have been perhaps less capable and harder to change. For backward compatability, the

comma was made optional.

The `break` command's syntax is `break` *loc [line] [*`when` *predicate].* Ignore the *predicate* for the moment. If *loc* is a procedure, then a breakpoint is set at its first instruction or at the specified *line*. If *loc* is a module, its initialization procedure is used. If *loc*'s type is compatible with `System.Address`, a breakpoint is set at the instruction at that address. This sort of polymorphism is common to many commands, and, although cumbersome to enumerate, seems to work well in practice.

7 Control and Programmability

Loupe provides the typical set of mechanisms for inspecting and controlling program execution. You can browse the stack and examine stack frames. You can browse the set of threads and focus on particular threads. The program can be started and stopped interactively. It can be single-stepped by instruction or by statement or within a procedure or by procedure call or by procedure return. Breakpoints can be set at interesting locations.

(One problem with such mechanisms is their use in a concurrent Modula-2^+ environment. If you are inspecting the execution of some particular set of threads, it should be possible to choose whether the other threads are stopped or running. The uneven passage of real time is also a problem. Loupe's current mechanisms in this area are inadequate but have caused no real problems so far.)

A breakpoint can have an associated predicate; the breakpoint will fire only when the predicate is true. One can say, for instance, `break Writer.Prin when depth < 0`. One can also say `break Writer.Prin when !(print "depth =", depth; value FALSE)` to print the value of `depth` whenever `Writer.Prin` is entered, but not stop.

The stepping commands can also take predicates. The `singlestep` command (not abbreviated `ss`, since `showstack` was already abbreviated `ss`) steps by one machine instruction; one can say `singlestep until AT(8cc04H, INTEGER) = 0` to solve a low-level core-smash problem before embarking on a bicycle tour of the Sierras.

It is also possible to call user procedures from Loupe. Saying `print Time.Now()` calls `Time.Now`, which returns `<seconds = 509856375, microseconds = 250000>`, which Loupe prints. You can plan ahead and include useful debugging routines into the program, then call them when necessary. It is interesting to note that user procedures thus called can hit breakpoints or

other conditions that cause a return to Loupe, as in

```
<9> break Time.Now
<Time.Now, line 185 ("PROCEDURE Now(): T;")>
<10>  Time.Now().seconds
running...
...debugger
<Time.Now, line 185 ("PROCEDURE Now(): T;")>:
stopped at breakpoint
<11> unbreak Now
<12> stepup
running...
...debugger
returned <seconds = 509856375, microseconds = 250000>
back to <10>
509856375
```

Here, the `stepup` command printed the value returned by `Time.Now`. Loupe then realized that it could resume command 10, and did so, notifying the user. Explicit notification is important because multiple procedure calls could be outstanding, and because, in the presence of coroutines or multiple threads, they could return in an unexpected order.

Procedure-call is of course dangerous; calling an arbitrary procedure from an arbitrary situation could violate various invariants. Loupe makes absolutely no attempt to protect the user from such mistakes. It could detect some potential problems, but might thereby lull the user into a false sense of security.

Another dangerous mechanism is provided by the `RETURN` command (whose name is upper-case for this reason). `RETURN` *[value]* forces a return from the selected procedure activation (which need not be at the top of the stack). One might imagine that Loupe would perform any appropriate finalizations as part of the `RETURN` operation, by analogy with exception-raising. It does not; the rationale is that programmers provide finalizations only for those cases that they believe could happen, based on their knowledge of the program structure; they would not necessarily have planned for an abnormal return initiated interactively. Since Loupe cannot solve the whole problem, it attempts no part of the problem, although its other facilities should allow the user to solve the problem by hand.

Loupe itself provides a number of programming features. To start, you can bind global names using the `define` command; `define` *name* = *value*

binds *name* to *value* in the global scope. To rebind a global name, `DEFINE` is used (upper-case for safety; one can say `DEFINE FALSE = TRUE` and Loupe will do it). A similar feature provides a history mechanism; the global name `$` is always bound to the result of the most recent command (that had a result), and `$`*n* is bound to the result of command *n* (if any).

(Loupe's model of sharing is different from Modula-2$^+$'s, as seen in

```
<13>  x
5
<14>  LOCOF(x)
<the 32 bits at <loc 05d740H>>
<15>  LOCOF($13)
<the 32 bits at <loc 05d740H>>
<16>  x := 4
<17>  $13
4
```

This model is convenient for interactive use; instead of saying `x := 4`, one could have said `$13 := 4`. One can also have said `AT(5d740h, INTEGER) := 0`, which is often handy. For cases where such sharing is undesirable, the built-in procedure `COPY(`*x*`)` returns a copy of *x* that does not share storage with it.)

Loupe provides a number of commands for interactive programmability. The `lambda` command returns a lambda-expression.

```
<18> define floattime = !(lambda (time)
     FLOAT(time.seconds) +
     FLOAT(time.microseconds) / 1000000.0)
<19>  floattime
<<lambda (<<"time">>)
 <<<"+">>(
    <<<"FLOAT">>(
       <<<".">>(
          <<"time">>, <<"seconds">>)>)>,
    <<<"/">>(
       <<<"FLOAT">>(
          <<<".">>(
             <<"time">>,
             <<"microseconds">>)>)>,
       <1000000.000000>)>)>>
```

```
<20>  floattime(Time.Now())
running...
...debugger
509860960.000000
```

(As a syntactic variation, one can phrase the above as `define floattime(time) = ....`) Similarly, the `let` and `letrec` commands provide syntactic sugar for lambda-binding, and the `LET` command changes the value in an existing binding (like Lisp's `setq`).

Lambda-bindings are lexically scoped and take precedence over Modula-2^+ bindings. (Getting such a feature right can be tricky. Loupe once happened not to allow `let oldDepth = COPY(Writer.depth) in !(break Writer.Prin when depth # oldDepth)` because the `when` predicate was not properly closed in the lexical context.) Loupe's provision of lambda-evaluation guarantees tail-recursion, eliminating the need for other control structures.

Even so, Loupe also provides the more conventional `for`, `while`, and `repeat` commands, which are more familiar to Modula-2^+ programmers. (In the `for` command, the controlled variable is lexically bound; its type is deduced from the types of the bounds. As a result of the Modula-2^+ type system, `UNSIGNED` iteration cannot be supported in this way.)

8 Interaction with the Compiler

Loupe's model of the program being debugged, and of the Modula-2^+ language, should be as close as possible to the compiler's.

Loupe bases its model of the program being debugged on symbol-table information received from the compiler. The compiler produces a Unix-style object file for each module compiled, and passes symbolic information to the debugger by encoding it in cryptic ASCII strings in the object file. The loader coalesces these strings, uninterpreted, into the symbol table of the executable program. Unix defines standard ASCII encodings for symbolic information from C and Pascal programs; DEC's Western Research Laboratory's Modula-2 compiler implemented additional encodings for Modula-2; we improved and extended these for Modula-2^+.

The amount of symbol-table information present in the executable file is quite large; each module's object file contains, for example, the definitions of all the types defined there, plus all the types imported directly or indirectly. Popular types are thus defined in each module that imports them, and the

loader preserves all definitions. (Naturally, the definitions are represented in such a way as to allow Loupe to realize that they are all for the same type.) Since Loupe parses the symbol table into an internal form, this mechanism has proved cumbersome. Early versions of Loupe parsed the entire symbol table at once, but this was far too slow on large programs. Loupe now uses a complicated lazy-parsing scheme to help achieve adequate performance.

Since Loupe's only information about the static structure of the program comes from the symbol table, it is important that it be as complete and accurate as possible. Unfortunately, the complex symbol-table structures have become difficult to extend. For example, we recently improved our optimizer to perform lifetime analysis of local variables and common subexpressions; the optimizer can detect variables with disjoint lifetimes and place them in the same registers or locations. Unfortunately, there is no current way for the optimizer to pass this information along to Loupe, and there are practical obstacles to providing one. Since programs compiled with lifetime analysis could not be reliably debugged, much less easily, lifetime analysis has not been released to users.

As a smaller example, the symbol-table syntax for compile-time constants defined by `CONST` is based on the various types such constants can take on. A compiler change had the effect of increasing the range of types to include `System.Address`, but the symbol table syntax could not easily be extended. As a result, `System.Address` constants seem to Loupe to have `INTEGER` type.

Our ultimate plan is to implement symbol tables using a highly efficient persistent-data mechanism now under development. This will allow compiler data structures, defined using the Modula-2^+ type system, to be passed (in a suitably edited form) straight to Loupe, and should better allow for future extensions while eliminating the need for the compiler to generate and for Loupe to parse ASCII representations.

Implementing Loupe has also helped to refine the compiler and the language. In the early days of Modula-2^+, it was the compiler that best defined the language. A language change required a compiler change, but a compiler change *was* a language change. Since implementing Loupe required reimplementing a large part of Modula-2^+, this helped to refine the definition of the language.

For example, it was nowhere documented exactly which types the various Modula-2^+ operators and built-in procedures would take, and what types they would return. Exhaustive testing provided the results, some of which were surprising; for example, the constant `NIL` was considered type-

compatible with INTEGER. The non-surprising results were implemented in Loupe; the surprising ones were marked to be fixed in the compiler and implemented correctly in Loupe.

9 Interaction with the Runtime System

Loupe also needs to understand the runtime environment presented by Modula-2^+ and by the commonly used packages. Loupe does a poorer job here than elsewhere.

Modula-2^+'s runtime system provides support for language constructs as well as a rich and evolving set of commonly-used packages. The runtime system, for example, provides reference counting as a mechanism for garbage collection. Loupe ought to interact well with reference counting, but does not yet. Setting the value of a reference-counted variable, for example, should update the reference counts of the old and new referents. Such updating is delicate since the program may be in the middle of making such an update itself. Similarly, if Loupe makes a copy or discards a copy of a reference-counted variable, it should update the reference count of the referent.

In general, any Loupe mechanism involving detailed, automatic manipulation of the program state is difficult to provide. Loupe does not provide the built-in procedure NEW, since the program's runtime allocator could be in an arbitrary state when Loupe's NEW was called. Providing NEW in Loupe would require tighter coupling with the runtime allocator, and would make it more difficult to experiment with different allocators.

Loupe does not understand Modula-2^+ exceptions as well as it should. For example, the **stepup** command steps up from the selected procedure activation by setting a special breakpoint at the point of return, then running the program until the breakpoint fires, then removing the breakpoint. If an exception causes the procedure to return abnormally, the breakpoint will not fire and the program will continue to run. Of course, a clever user could set a breakpoint on the procedure `Signaller.Raise`, which implements RAISE, but Loupe should be clever instead.

Similarly, the multi-threaded runtime environment contains a wealth of information available raw to the experienced user, but confusing to others. Loupe provides some information directly in a friendly way (*e.g.,* whenever Loupe gains control from a thread different from the previous thread, Loupe notifies you of the new thread); it could do more.

Loupe also incorporates some knowledge of commonly-used packages. An example is the Text package, which provides operations on garbage-collected byte strings. Most Modula-2+ programs use Text objects at some level, but only the implementers of the Text package should be expected to understand its implementation. Loupe therefore prints a Text object (which is a REF to a variant record) as a character string; the internal representation of a Text object can be examined by dereferencing it. Although Modula-2+ allows Text literals, Loupe cannot, since this would involve calling NEW.

As the number of widely used types grows, Loupe will have a harder time keeping up. One long-term solution would be for Loupe to provide mechanisms to allow the implementors of the various types to provide the necessary debugging support themselves. For example, the Cedar system developed at Xerox PARC allowed a PrintProc ("print procedure") to be associated with a type; the PrintProc was called to print a value of that type. Until such systematic facilities are provided, Loupe's users can call such procedures manually if they exist.

10 Retargetability

Loupe is used to debug programs under Unix and under the Topaz software system on the Firefly multi-microprocessor computer. Different versions of the Firefly have used different processor families.

Loupe is therefore retargetable, both across software environments and across processor architectures. Most of Loupe is independent of the target environment; a target-specific part is provided for each target.

On Unix, Loupe allows debugging a subprocess (using Unix's *ptrace* mechanism) or post-mortem debugging from a core file. Loupe also supports teledebugging a Topaz program from another machine, either Unix or Topaz, or from the same machine. In the teledebugging case, the program need not have been started from under Loupe. Topaz core files can also be debugged.

The Topaz software structure provides multiple address spaces, each with multiple threads of control. As a special case, the lowest-level *Nub*, which provides address spaces and threads of control, can also be teledebugged (from another machine, of course). Loupe's unit of debugging is a single address space, which is a convenient unit in practice and happens to be the unit across which Modula-2+ names are consistent. If you wish to debug multiple address spaces, you must use a separate instantiation of Loupe for

each. Since these Loupes cannot communicate, you cannot automatically manage data or events across the address spaces, but this lack has not yet proved a practical problem.

There are obvious major differences in Loupe support for the various targets, and many small ones. For example, Loupe often invisibly dereferences `POINTER` values to check their validity; it does this by attempting to fetch the first and last bytes of the referents. This is quite dangerous if the referent is a device register. Loupe's rules for `POINTER` validation are thus different for the different targets.

Loupe's support for different processor families centers on their differing data representations. Loupe models memory as being bit-addressed, and has per-processor rules for mapping data items onto bit addresses; these rules correct for different byte orders. A very small amount of code that perform actual data access translates between bit addresses and machine addresses; this allows easy, centralized support for teledebugging between different processor families. (Early implementations of this mechanism were quite inefficient but general; the inefficiency was unimportant since the rate of memory access was ultimately limited by the user's speed of typing. With the addition of programmability to Loupe, it was necessary to speed up this mechanism; this speedup was achieved without loss of generality.)

11 Acknowledgements

Loupe was implemented by the author and David Redell. The Modula-2^+ compiler was provided by Paul Rovner, Violetta Cavalli-Sforza and Christine Hanna; it was based on the Modula-2 compiler written by Michael Powell at DEC's Western Research Laboratory.

Loupe was written in Modula-2^+ and debugged using Loupe.