Technical Report

UCAM-CL-TR-161
ISSN 1476-2986

Number 161

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Papers on Poly/ML

D.C.J. Matthews

February 1989

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Papers on Poly/ML

D.C.J. Matthews

Preface.

Various papers and research documents have been written while the Poly/ML project was under way and now that the project is completed it seemed appropriate to combine them into a single document. Some of these papers have been published, while others were written simply to help clarify thoughts on a particular subject.

Poly/ML started in 1983 when Dr Mike Gordon suggested that I should try to implement the Standard ML language which was then being designed by a group led by Prof. Robin Milner. At that stage the core of the language had been more-or-less fixed though there was still no facility for modules.

I had finished a compiler for Poly written in Poly and had it running on a Vax. The compiler consisted of four sections. The first part was a parser and type-checker for Poly. The second section took the Poly parse tree and generated an intermediate code tree structure. The third phase was an optimiser which worked on the code tree, and in particular expanded inline functions, and the final section was a code generator producing the Vax machine-code. The whole compiler ran under a persistent storage system.

Although Poly and ML have different syntax and type-systems their execution models are remarkably similar. A new parser and type-checker had to be written but the code-generator and optimiser could be shared between Poly and ML. The same run-time system was used.

By the end of 1983 an experimental version of Poly/ML was running and being tested in the Computer Laboratory by Dr Larry Paulson and by Prof. Mike Fourman at Brunel. Subsequently there has been work in a number of areas. The ML language has changed and in particular a mechanism for modules has been added. The system has been ported to the Sun computer which necessitated a new code-generator and substantial changes to the run-time system. Recently a window system and parallel processes have been added.

What lessons have been learnt from the project? Poly turned out to be an excellent language for the project. Modules based around Poly abstract types were used extensively. The lack of the kind of

low-level "bit-twiddling" operations of a language like C was not a problem even in the machine-code generators. In practice compilers involve very few such low-level operations. Polymorphic operations were used extensively. Occasionally it would have been nice to have been able to use polytypes as in ML rather than having to explicitly declare instances of types, however there was at least one case where a polymorphic function was written in Poly which could not have been written in ML (because a polymorphic function in ML cannot contain a recursive application to a specific type).

The papers in this report have been grouped into sections according to their likely audience. The first section describes the Poly/ML system and the extensions for windows and processes. This section is likely to be of most interest to users of Poly/ML.

Section 2 contains various discussion papers about Poly and ML. Some of the ideas covered were never actually implemented, in particular the work on structure editing.

The third section contains two papers on the persistent storage system and its implementation.

The final section covers the implementation of Poly and ML and the run time system. The papers in this section describe the internals of the compiler and the run-time system.

1. **Documentation for Poly/ML.**
   a. The Poly and Poly/ML Distribution.
   b. Processes for Poly and ML.
   c. Windows for ML.

2. **The Design of Poly and ML.**
   a. Exceptions with Parameters for Poly.
   b. Structure Editing in Poly and ML.
   c. Static and Dynamic Type-Checking.

3. **Persistent Storage.**
   a. A Persistent Storage System for Poly and ML.
   b. Progress with Persistence in Poly and Poly/ML.

4. **The Implementation of Poly and Poly/ML.**
   a. An Implementation of Standard ML in Poly.
   b. The Poly and ML System: Abstract Machine and Realisation.
   c. Machine Independent Optimisation in Poly and Poly/ML.
   d. Hardware Support for Poly and ML.
   e. The Poly and ML System: Low Level Support.
   f. Interfacing C Routines to Poly/ML.

The Poly and Poly/ML Distribution

David C.J. Matthews

20 May 1988

1 Getting Started

The Poly and Poly/ML distribution tape contains the basic system for running Poly and Standard ML. There are three files on the tape

- driver
- poly_dbase
- READ_ME

To build the system you need to read off the directory **driver** and the heap image **poly_dbase**. The driver must be made using the standard Unix **make** program specifying the system you are building it for. To build a system for the Vax you need to give the command line

```
make S = vax
```

for the Sun3

```
make S = sun
```

and for the portable version

```
make S = int
```

The result will include the driver program **poly** and the disc garbage collector **discgarb**. These should be moved to a convenient place. Note that the heap image also contains machine code and so you have to have different heap images for the Sun and the Vax. You can now run the system by giving the command

```
poly poly_dbase
```

You will now get a message from the system and finally a prompt from Poly. The system comes up in Poly rather than ML by default. It is possible to change this and how to do it is described below. In either case the mode of working is basically the same. You can type expressions or declarations which will be compiled and executed. Each declaration is added to the environment. You finish your session

by typing the end-of-text character (usually ^D). All the new declarations will be written onto the database so that they are available next time.

The database contains the Poly and ML compilers and everything else. Each user must have his or her own copy of **poly_dbase** because any new declarations made in a session are added to it[1]. It is possible for several people to share the same database provided they do not want to write back to it. In that case the database should be made read-only using the Unix **chmod** command or alternatively the `-r` option can be given on the command line.

```
poly -r poly_dbase
```

In both cases changes or new declarations will not be written back at the end of the session.

2 Poly

The Poly compiler implements a slightly revised version of the language as described in the Poly Manual[2].

The main difference is that declarations can be combined in a single declaration introduced by **let** or **letrec** and separated by **and**. This allows mutually recursive declarations, for example

```
letrec f == proc ... and g == proc ...
```

The other differences are that the **prefix** keyword has been removed and additional keywords **infixr**, **cand** and **cor** have been added as well the **and** keyword already mentioned. **infixr** can be used instead of **infix** for identifiers with right-associativity. **cand** and **cor** behave syntactically like infix operators of precedence -1 and -2 respectively and provide short-circuit evaluation of **boolean$&&** and **boolean$||** .

Some other procedures and types have been added since the Manual was produced.

2.1 ml

The **ml** procedure enters the ML system interactively. To return to Poly type end-of-text (^D). To set up the system to run ML instead of Poly by default there is a procedure **install_ml** which can be called. Next time the database is used it will be in the ML rather than Poly system. A similar procedure **install_poly** will set the system to run Poly rather than ML.

---

[1] But see **make_database** below

[2] Technical Report 63. University of Cambridge Computer Laboratory.

2.2 long_integer

long_integer is now an arbitrary precision integer type, so **first** and **last** have been removed, and only **to_integer** raises the **range_error** exception.

2.3 commit

The **commit** procedure now writes out local declarations to the database without terminating the program so it can be called at any time to update the database. It will raise `commit_failed` if the database cannot be written to or if the `-r` option was given.

2.4 quit

quit leaves the Poly system without writing back any changes made. It never returns.

2.5 real

There is a type **real** which implements real numbers.

2.6 # and make

The # procedure which compiles a text file now tries to find a file name with the extension `.poly` if the file name given does not exist. Similarly the **make** procedure will look for files with the extension if it does not find a file with the required name. This does not apply to directories or `poly_bind` files.

3 ML

The version of ML is an almost complete implementation of the standard defined in the October issue of "Polymorphism" and includes the `BasicIO` structure and the modules proposal. The `PolyML` structure contains some non-standard functions which can be called from ML. They can be used directly by using the name prefixed by `PolyML.` or by declaring the function in the global environment.

```
val use = PolyML.use; use "myfile";
```

You can of course choose your own name.

```
val load = PolyML.use; load "myfile";
```

It is best not to try open `PolyML` unless you are sure you will want all the declarations from it with the names that are given.

3.1 commit

`PolyML.commit: unit -> unit`
commit writes out any new declarations to the database.

3.2 quit

`PolyML.quit: unit -> unit`
quit leaves the system without writing back any changes.

3.3 poly

`PolyML.poly: unit -> unit`
The function **poly** enters the Poly system. Typing end-of-text (^D) will return to ML.

3.4 use

`PolyML.use: string -> unit`
The function **use** takes a file name as a parameter and compiles it as ML program text. If the file does not exist it tries for a file with the extension `.ML`.

3.5 print_depth

`PolyML.print_depth: int -> unit`
The argument to **print_depth** controls the depth to which structures are printed either with `print` or when printing the result of an expression or declaration. It can be useful to avoid printing large lists when they are produced as top-level results.

3.6 timing

`PolyML.timing: bool -> unit`
The **timing** function takes a boolean argument and switches on or off timing information.

3.7 profiling

`PolyML.profiling: int -> unit`
It is sometimes useful when debugging a program to have some idea of where it is spending most of its time. The **profiling** function can be used to switch on and off profiling of ML code. Switching on profiling causes the system to print out a list of procedures and counts. The value printed depends on the statistic being measured and this is controlled by the value of the argument given to **profiling**. Possible values are

- 0 - Switch off profiling.
- 1 - Give the time spent in each procedure.
- 2 - Count the number of words of store allocated by the procedure.
- 3 - Count the number of persistent store traps raised from each procedure.

The most useful of these is likely to be the time statistic. The other two are mostly of use only for debugging the system.

3.8 make

`PolyML.make: string -> unit`

The **make** function in ML is basically the same as **make** in Poly and performs a similar function to its namesake in the Unix system. It is concerned with re-compiling a program from a set of modules in a way that does not require more recompilation than necessary. Unlike the Unix **make** program it uses normal ML statements to express the dependencies between modules rather than having a separate language.

A large program is normally made up of a set of modules which are combined together to make it. In ML this is done by applying functors to structures to make higher-level structures which are in turn used as arguments to other functors until the highest-level structure is reached which is the program. This can be thought of as a tree structure in which the functors and the basic structures are the leaves and the intermediate structures are branches. The root of the tree is the final program. The intermediate structures are constructed by simple declarations such as `structure S = F(A,B);` where `A` and `B` may be leaves or other intermediate structures.

In addition to describing how to make `S` from `F`, `A` and `B` this declaration also says that `S` depends on `F`, `A` and `B` in the sense that if the source code for `A`, say, has been changed we would have to recompile it and make a new `S` by re-executing the declaration. The **make** function uses these declarations both to decide on the dependencies and to remake the modules.

make needs to be able to find the source code for each module to decide if it has been changed. To use it the source files making up the program must be set up in the filing system in a way which reflects the tree structure of the program. Intermediate nodes are represented by directories and leaves by text files. So, if we have a structure `S` which depends on a functor `F` and other structures `A` and `B` we must put the sources of `F`, `A` and `B` in text files `F.ML`, `A.ML` and `B.ML` together in a directory called `S`. The declaration which binds these together `structure S = F(A,B);` goes in a file called `ml_bind.ML`. If you now type `make "S";` the function will look at `S/ml_bind.ML` to find the dependencies. It will find out when each structure or functor was last made and when the corresponding source file was changed, and recompile it if the source file is newer. If it has had to bring any of

the structures or the functor up-to-date it will execute the code in ml_bind.ML to make a new S.

This can be used recursively, so that if A, say is made up from a functor application rather than being a leaf structure it will be represented by a directory called A inside the S directory rather than a source file A.ML. The directory will contain a ml_bind.ML containing a declaration of a structure A.

Programs are not always tree structures and sometimes a structure is shared between two other structures. If A and B were both intermediate structures referring to a common structure D. Then A/ml_bind.ML would contain something like structure A = G(D); and B/ml_bind.ML might contain structure B = H(D); . The source code for G would be in S/A/G.ML and that for H in S/B/H.ML but that for D is needed in both S/A and S/B. The solution is to put the code at some point on the branch of the tree that contains both A and B, i.e. in S/D.ML. The **make** function knows the path by which it reached a directory and will search for a source file back up this path. Even though D is mentioned in two different places it will be made once so the same structure will be used in both cases.

3.9 make_database

PolyML.make_database: string * string -> unit

This version of the Poly/ML system allows several users to share ML values, such as functions or structures, by having access to a shared database. **make_database** can be used to create new databases for users who can then share the declarations in the parent, and if required, in each other.

The basic principle is that each user has their own database which is the one they give on the command line to poly. Any new declarations they make go into here when they commit. They can make use of values in other databases provided there is some sort of path between their own database and the one they want to use. They can read values and assign to references in other databases but changes will not be written back and will not persist after the end of the session.

While the underlying system allows an arbitrary set of references between databases to be set up the mechanism for creating new databases in Poly/ML makes databases in a heirarchy. A user of a database further down the heirarchy can make use of declarations in any of its parents. Suppose that the database you are currently using is in a file called poly_dbase, i.e. you gave the command poly poly_dbase when you started. make_database("Fred", "freds_dbase"); will create a database in a file called freds_dbase. This is a *child* of poly_dbase. You can now use the child by giving the command poly freds_dbase and use any of the declarations you previously had in poly_dbase and any of its parents. Any new declarations you make will go into freds_dbase when you finish or use commit. You or someone else can still run the system by giving the command poly poly_dbase and add new declarations which will be available there or to the user of freds_dbase. The new declarations in freds_dbase are not available directly to the user of poly_dbase but are contained in a structure Fred and can

be used either by open `Fred;` which will make available the *current* contents of that database or by using the dot notation e.g. `Fred.it` will be the last expression evaluated.

There are a few restrictions on the way that multiple databases can be used. Each database contains the names of the other databases to which it may refer. Since these are file system names it is possible to confuse the system fairly easily and this essentially means that once a database is created it should not be moved.

The driver program assumes that databases will not change under its feet so it obtains a lock on a database that it wants to use. Several users are allowed to read from a database at once but only one is allowed to have it open for writing. A lock is obtained on the database given on the command line when Poly/ML starts up but other databases are only locked when something is read from them. In practice this means that using a database further down the hierarchy will cause all the databases above it to be locked, so a parent can only be opened for writing if no-one is using any of its children. This suggests that shared databases should be set up to contain declarations which several users will need but which will not be changed frequently. An obvious example is the database containing the Poly/ML system. Other databases in the heirarchy will be locked if something from that database is used.

If the users do not require access to each other's declarations but only to declarations in the shared database you can create a single database for a canonical user and then make copies of that for each individual user. In this case the database should be created using a command such as `make_database("User", "user_dbase");`. Each user of copies of `user_dbase` will see their own declarations in the `User` structure. This has the advantage that it is impossible for a user of one copy of the database to lock another since their is no way to make use of anything in any of the other copies.

4 discgarb

New declarations are added to the database at the end of each session or when **commit** is called from Poly or ML. If a previous declaration is overridden by a new declaration with the same name the old object will become inaccessible, though it will still be on disc. The tape contains a program **discgarb** which can be run on a database to remove inaccessible data.

```
discgarb poly_dbase
```

Running **discgarb** like this will not make the database any smaller but will make the space available so that the next time objects are written out the old space will be used before the database is extended.

It is also possible to give one of two options to **discgarb** before the name of the database. The `-r` option prints the amount of space that would be freed but does not actually change the database. The `-c` option compacts the database to

reduce the disc space used. To avoid corrupting the database it does not remove all the free space at once so it may have to be run two or three times to reduce the file to its minimum size.

5 Interrupt Handling

A Poly or ML program can be interrupted by typing control-C (or the appropriate interrupt character on your machine). It will then produce a prompt (=>) and will expect a single character followed by a carriage-return. The possible commands are:

- q - exit from the poly system without writing to the database.
- c - continue from the interrupt.
- t - give a trace of the active procedures.
- f - raise an exception.
- ? - give a list of these commands.

6 Problems and More Information

Much of the aim of distributing the system is so that I can get feed-back from users. I would much appreciate comments, problems, bug-reports or suggestions. I will distribute any bug-fixes to those who are actively using either Poly or Poly/ML so please let me know how you get on.

Dave Matthews
Computer Laboratory,
University of Cambridge,
Corn Exchange Street,
Cambridge CB2 3QG
England.

Arpa: < @ucl-cs: dcjm@cl.cam.ac.uk >
Janet: dcjm@UK.AC.Cam.CL
UUCP: mcvax!ukc!cl-jenny!dcjm
+44 223 334632

Processes for Poly and ML

David C.J. Matthews

October 12, 1987

This document describes the process mechanism as it is currently implemented in Poly and ML. Though implementing processes in functional languages is interesting in itself the main motivation was the need to be able to handle asynchronous activities in a window system.

The system is largely based on the ideas of CCS [2] and the PFL implementation [1]. The main difference is in the *choice* function.

1 Channels

Poly

```
channel:
   PROC(t: TYPE END)
      TYPE
      assign: PROC(t);
      content: PROC()t
      END
```

ML

```
type 'a channel
val channel: unit -> 'a channel
val send:    'a channel * 'a -> unit
val receive: 'a channel -> 'a
```

Processes communicate over *channels* which are unbuffered. When a process wishes to send data it must wait until a second process is ready to receive it. The

data is transferred and both processes can proceed. Similarly a process wanting to receive is blocked until a process is prepared to send.

In both Poly and ML channels are created by a call to *channel*. The result however is slightly different because of differences in the type systems. In Poly a channel behaves exactly like a variable since it is a **type** with *assign* and *content* procedures. Values are sent by assigning to the channel and received by taking its value.

The type of a channel in ML is more complicated. There is a type, also called *channel*, which is the type of values returned from the *channel* function. Values are sent and received using the *send* and *receive* functions. A problem with this is that a channel is very similar to a reference and must be constrained to be a monotype if the security of the type system is to be preserved. This means there is a restriction that the result of the *channel* function must be a monotype which will almost certainly require an explicit type constraint.

2 Processes

Poly

```
process:
   TYPE
   fork: PROC(PROC());
   console: PROC(PROC())PROC();
   choice: PROC(PROC(); PROC())
   END
```

ML

```
val fork:    (unit->unit)->unit
val console: (unit->unit)->(unit->unit)
val choice:  (unit->unit)*(unit->unit)->unit
```

The simplest way of starting a process is with *fork*. This takes a procedure as its argument and runs it as a separate process. The process calling *fork* continues unaffected.

console is similar to *fork* but creates a process which will receive an exception if a console interrupt is raised. It also returns a procedure as its result which can be called to simulate a console interrupt in that process only. This provides a way of interrupting programs which have gone out of control.

choice forks two processes and returns immediately. However unlike calls to *fork* only one of the processes is allowed to send or receive on a channel. If both processes try to send or receive only one of them will be successful and the other will be blocked for ever. If the first action of both processes is to receive or send on a channel then *choice* behaves exactly like the choice primitive of CCS. Either of the processes created by *choice* can create more processes before sending or receiving. In that case the choice is between the processes created by the two alternatives. If any of the processes created by one alternative makes a successful transfer (either sends or receives data) then none of the processes created by the other alternative will be able to. In this way *choice* is associative and commutative since if the first action on either alternative is either to send or receive or to create a process with *choice* which behaves in this way, then only one of the choices will be taken, and the order of the choices is irrelevent.

References

[1] S. Holmström, *PFL: A Functional Language for Parallel Programming*, Proceedings of the Declarative Programming Workshop, University College London, 1983.

[2] R. Milner, *A Calculus for Communicating Systems*, Lecture Notes in Computer Science, no 81, Springer Verlag, 1981.

Windows for ML.
Revised.

David C.J. Matthews

16 December 1987

This revised version of the windows system incorporates various changes and suggestions. It is still open to change but I hope it will soon converge on a standard. Any suggestions are welcome.

The main aim is that it should be reasonably portable while allowing ML programmers to do most of what they want directly in ML. It has been implemented on the Sun but there may have to be a few changes for other implementations. An implementation under X-windows is probably the next step.

The design generally follows the **Bitmaps Proposal** of Mike Fourman who in turn based it on bitmaps in Luca Cardelli's language, Amber. It has been revised in the light of suggestions from Mike Fourman and others at Abstract Hardware and IST.

1 Figures

```
structure Figures :
    sig
    datatype point = point of {x: int, y: int}
    val origin : point
    val add_pt : point * point -> point
    val sub_pt : point * point -> point
    val scale_pt : point * int -> point

    datatype line = line of point * point

    datatype rect =
       rect of {bottom: int, left: int, right: int, top: int}
```

```
    (* Points in the rectangle *)
    val left : rect -> int and right : rect -> int
    val top : rect -> int and bottom : rect -> int
    (* Height and width *)
    val height : rect -> int and width : rect -> int

    (* create_rect - make a rectangle enclosing two points.
       The pixel in the bottom left is in the rectangle,
       the pixel in the top right is not. *)
    val create_rect : point * point -> rect
    val top_left : rect -> point
    val top_right : rect -> point
    val bottom_left : rect -> point
    val bottom_right : rect -> point

    val left_of : rect * rect -> bool
    val above : rect * rect -> bool
    val sects : rect * rect -> bool (* Rectangles intersect *)
    val offset : rect * point -> rect (* Move a rectangle *)
    val size : rect -> rect (* Moved to origin *)
    val union : rect * rect -> rect (* Surrounding rectangle. *)
    val intersection : rect * rect -> rect
    exception intersection : unit
    val within : point * rect -> bool
    end
```

The Figures structure contains types implementing points, lines and rectangles, and operations on them. The origin is assumed to be at the lower left hand side of the screen so rectangles are created with the value of bottom less than or equal to top and left less than or equal to right. To allow for empty rectangles the bottom left point is assumed to be in the rectangle but the top right is not.

2 Bitmap

```
structure Bitmap:
   sig
   type bitmap and mode

   val create_bm: Figures.rect -> bitmap

   exception bitmap: string
```

```
val draw:
   {source: bitmap, dest: bitmap, from: Figures.point,
    to: Figures.rect, draw_op: mode} -> unit
val fill:
   {dest: bitmap, to: Figures.rect, draw_op: mode} -> unit
val draw_line:
   {dest: bitmap, line: Figures.line, draw_op: mode} -> unit
val draw_pattern:
   {source: bitmap, dest: bitmap, from: Figures.point,
    to: Figures.rect, draw_op: mode} -> unit
val region: bitmap * Figures.point * Figures.rect -> bitmap
val bounds: bitmap -> Figures.rect
val init_bitmap: bitmap * int -> unit

datatype font = font of int
val draw_text:
   {message: string, dest: bitmap, font: font,
    where: Figures.point, draw_op: mode} -> unit
val font_default : font
val char_box: font -> Figures.rect

val mode_src: mode and mode_dst: mode
val mode_set: mode and mode_clear: mode
val mode_not: mode -> mode
val mode_or: mode * mode -> mode
val mode_and: mode * mode -> mode
val mode_xor: mode * mode -> mode

(* The kind of access allowed. *)
val readable : bitmap -> bool
val writeable : bitmap -> bool
val freeze: bitmap -> unit
end
```

The Bitmap structure contains operations for operating on bitmaps. Bitmaps are rectangular areas of bits which can represent pixels. There are two kinds of bitmap: Those which are associated with areas of the screen which can only be written to, and bitmaps in memory which can be read from or written to.

The bounding box for a bitmap, returned by bounds, gives the height and width and also an origin for drawing operations on the bitmap.

A bitmap of a given size and origin is created with create_bm. This creates a new

bitmap in memory initialised to zeros. A bitmap can also be made from an area of an existing bitmap. `region` creates a bitmap which shares the same bits but with a new origin and rectangle. The point is the origin of the region relative to the parent, and the rectangle is the area of the region. The region must lie entirely within the parent. Drawing operations on a region are clipped to its edges. As well as providing a clipping window a region can also be used to shift the origin of a bitmap without changing its size.

The primitive drawing operation is `draw` which copies from an area of one bitmap to an area of another. `from` is a point relative to the `source` and to a region in the `dest`. The way in which the copying is done is controlled by the `mode`. Each bit in the destination is set to a value formed from the corresponding source bit and the previous value of the destination bit by application of the appropriate function.

`fill` can be used to fill an area. It is the same as `draw` with a source which is all zeros.

`draw_line` draws a line using the appropriate drawing operation. The line is clipped to the region of the bitmap.

`draw_pattern` fills an area of the destination with copies of the source laid side-by-side and end-to-end. The `from` point allows the pattern to be aligned in the destination.

`draw_text` writes a string in a given font. The fonts are identified by integers and at present there is only one. `where` gives the location of the *origin* of the first character in the string. The y co-ordinate is the position of the base-line for the characters so characters with descenders will go below this line. All characters in a font must have the same width and the size of a character can be found with `char_box`. The rectangle returned is the size of a character and is located relative to the base-line. Its top line will have a positive y co-ordinate, being above the base-line, and its bottom line will have a negative y co-ordinate to allow for descenders.

`init_bitmap` allows a bitmap to be filled in with bits from a number so that a bitmap produced externally can be read in.

It is useful to be able to freeze the data in a bitmap in memory once it has been created so that no changes can be made to it. `freeze` can be used to do this. Once frozen a bitmap can only be used in the source of a drawing operation.

3 Windows

```
structure Window:
```

```
sig
type window
(* Get a default value for the shape of a window. *)
val next_win_posn : unit -> Figures.point * Figures.rect

(* Create a top-level window. *)
val create_win : Figures.point * Figures.rect -> window
(* Create a sub-window *)
val sub_win : window * Figures.point * Figures.rect -> window
(* Get rid of a window and any sub-windows. *)
val destroy : window -> unit

(* Get and set the size and position of the window. *)
val get_origin : window -> Figures.point
val set_origin : window * Figures.point -> unit
val get_bounds : window -> Figures.rect
val set_bounds : window * Figures.rect -> unit

.....

end
```

The Window structure is the most complex part of the windows package. Windows are thought of as forming tree structures with top-level windows as roots. Sub-windows can be formed from parts of other windows and moved around within them.

create_win and sub_win make windows with a given origin position relative to the parent, and a given rectangle relative to the origin. The position of the rectangle on the screen or in the parent can be changed, either by changing the origin or by changing the bounds. Changing the bounding rectangle can resize the window.

next_win_position returns a default position and rectangle for a new top-level window. Repeated calls may return different values so that windows stack up rather than completely covering one another.

```
structure Window:
   sig
   ....
   (* Get the current bitmap. *)
   val image : window->Bitmap.bitmap

   (* Cursor for the window. *)
```

```
    type cursor (* =  Bitmap.bitmap * Figures.point * mode *)
    val cursor_size : Figures.rect
    (* A default cursor. *)
    val arrow : cursor
    (* Set the cursor *)
    val set_cursor : window * cursor -> unit

    ...
    end
```

Drawing to the window is done by writing to the image bitmap of the window. The drawing takes effect immediately onto the area of the window but is not retained so if a window is moved on the screen it may be necessary to redraw some or all of the image.

Each window and sub-window has a cursor which is displayed when the mouse is in the window. This can be set by `set_cursor`. A cursor is a readable bitmap of size `cursor_size`, together with a hot-spot and a mode in which to draw it. The hot-spot is a point relative to the cursor which is treated as the actual cursor position.

```
structure Window:
    sig
    ....
    val is_top : window -> bool
    (* Set the name stripe *)
    val set_name : window * string -> unit

    (* Convert between open and iconic forms. *)
    val close_win : window -> unit
    val open_win : window -> unit
    val is_open : window -> bool
    (* An icon which can be drawn. *)
    val icon_size : Figures.rect
    val poly_icon : Bitmap.bitmap
    (* Set the icon for the window. *)
    val set_icon: window * Bitmap.bitmap -> unit
    ...
    end
```

Certain operations can only be performed on a top-level window. On the Sun a top-level window is created with a name stripe at the top and a border. Mouse

button clicks in this area can be used to ask for the window to be moved or resized. Other implementations might have resize or drag regions.

A top-level window can be closed into an inactive form in which it is represented by an icon. The icon is a 64 by 64 bit readable bitmap. poly_icon is a picture for the Poly system and is the default. The window can be made iconic using close_win and opened by open_win. Drawing to a closed window has no effect and changing its origin or bounds will only take effect when it is reopened. The system may allow the icon to be moved on the screen but this will not be reported to the ML program.

```
structure Window:
   sig
   ....
   (* Events which can be returned. *)
   datatype key_press =
      key_down | key_down_again | key_up | key_click
   datatype win_event =
      win_redraw of Figures.rect list | win_state |
      win_move of Figures.point | win_resize of Figures.rect |
      win_enter | win_exit | win_pointer of Figures.point |
      win_key of
         {key: string, location: Figures.point, state: key_press} |
      win_mouse of
         {button: int, location: Figures.point, state: key_press}

   type time_stamp (* Time when an event occurred. *)
   val before: time_stamp * time_stamp -> bool

   val read_win : window -> win_event*time_stamp (* Get an event *)

   (* Event set - An event will only be read if it is the mask for
      that window. Otherwise it will be passed on to the parent. *)
   datatype event_set =
       event_set_key | event_set_move | event_set_enter_exit |
       event_set_resize | event_set_pointer | event_set_state |
       event_set_mouse;
   (* Get and set the event mask. *)
   val get_mask : window -> event_set list
   val set_mask : window * event_set list -> unit
   ...
   end
```

Events may occur in the window and are read by read_win. A time stamp is

returned with an event to allow events on different windows to be put into the correct sequence.

Each window is a rectangular area and the ML program can draw over the whole of it. What appears on the screen may only be part of this because other windows may be overlapping it. The system clips all the drawing operations to the area which is actually exposed. If one of the overlapping windows is now moved to expose more of the ML window the ML program will have to fill in the new area. A `win_redraw` event indicates that the image should be redrawn. It includes a list of rectangles which should be filled in to completely redraw the image. The ML program can use this information to avoid redrawing the whole of a window if the cost of redrawing is high. This event is also generated if the window size is changed by a call to `set_bounds`.

The `win_move`, `win_redraw` and `win_state` events are all produced as a result of the user requesting changes to the window. On the Sun this is done using mouse clicks in the border of a top-level window. They are all treated as requests to the ML program to make the appropriate change, but it is up to the program to interpret and respond to the request. The user cannot force a particular action. The `win_move` event requests that the origin of the window be moved to the given point. The effect of setting the origin there will be to move the window without changing its size or affecting drawing within the window. The `win resize` event requests a change in the shape of the window. Responding by setting the bounds to the given rectangle will keep the origin at the same place on the screen but change the area being displayed. This response would be appropriate for graphics where one might want to look at a particular area while keeping the origin in the same place on the screen. In a window containing text it might be more appropriate to keep the origin at a fixed place within the window and so the correct response to a `win_resize` would be a combination of calls to `set_origin` and `set_bounds`. The `win_state` event requests the window to be opened or closed as appropriate.

The `win_enter` and `win_exit` events indicate that the mouse has entered or left the window. Within the window the position is returned by `win_pointer` events. The co-ordinates used are relative to the window. There is no way on the Sun to explicitly read the location of the mouse except by tracking events.

Key and mouse button presses are returned as `win_key` and `win_mouse` events. The buttons on the Sun mouse are numbered zero, one and two counting from the left. As well as the key or button pressed the position of the mouse at the time is returned. Some applications make use of repeated clicks of keys, particularly of the mouse buttons. `key_down` and `key_down_again` indicate that the key or button has been depressed and `key_up` and `key_click` that it has been released. `key_down_again` is given instead of `key_down` if that key had been released a "short" time before. Similarly `key_click` is given instead of `key_up` if a key is pressed and released quickly. In all cases when a key is pressed and released an

event will be generated for both the press and release. What constitutes a "short" time depends on the system.

Events are directed at the window which underlies the mouse, but a window can choose to mask out some or all events, in which case they will either be ignored, in the case of `win_enter` and `win_exit` or passed to the parent window, in the case of `win_pointer`, `win key` and `win_mouse`. `win_redraw` cannot be masked out. Which events a window is interested in is controlled by an event mask which can be read and set by `get_mask` and `set_mask`. Initially the mask is empty so all events are either ignored or passed to the parent. The set of events can be combined and events added to the mask or removed from it.

```
structure Window:
   sig
   ....
   val select_item_from_menu :
      window * string list * Figures.point -> int
   end
```

The final operation in the `Windows` structure allows a menu to be displayed and an item selected from it. The menu is a list of strings which are displayed and the number of the string selected is returned. The strings are counted from 1 and a result of 0 means that no item was selected. The menu is displayed at a position relative to the window. This will normally be the location of the mouse when a button is pressed to bring up the menu.

4 Discussion

There are a few outstanding questions which need to be thought about which may have a bearing on how easily the system can be ported.

A major change since the previous document is that redrawing a window after it has been uncovered is now the responsibility of the ML program. The previous version had a bitmap in memory for each window and all drawing was done to this. Redrawing the screen after damage was the responsibility of the system. It is still open to an implementation to do this in which case the `win_redraw` event would never be returned except when the window size is changed. The advantage of having the ML program redraw the window is that it may require less memory to hold a higher-level data structure than a complete bitmap. There is a trade-off in terms of CPU time but the ML program can always keep a memory bitmap and draw that to the screen each time.

A more difficult problem has to do with icons. On the Sun an icon is simply a window which represents a closed top-level window. It can be moved on the screen but it behaves like any other window, covering some windows and being covered by others. On the Macintosh however an icon has a more active role. Moving an icon into a window or onto another icon is interpreted as an operation involving both. A file is deleted by moving its icon to the icon representing a rubbish bin. This interpretation of an icon is difficult to implement on the Sun and probably other machines as well and so it has not been considered.

Sub-windows are areas within other windows which can be moved and resized within them. They are drawn separately from their parent window. They could be implemented using regions. Should they be primitive?

Menu selection is provided as a primitive operation. It could be implemented as, say, a sub-window, but that would probably be slower. Also the menu selection function as implemented can go over the boundary of the window. A more general function allowing pictures as well as strings is probably required.

Exceptions with Parameters in Poly

David C.J. Matthews

This document is a suggestion on how to put exceptions with parameters into Poly. The idea is essentially that suggested by Alan Mycroft for Standard ML.

1 Background

This can be skipped by those familiar with the ML exception mechanism.

If we want exceptions to have parameters we need to guarantee when we raise an exception with some parameters, that when we catch it, the parameters have the signatures we expect. A simple way of doing this is to introduce an exception declaration such as `exception x(string);`. This exception can be raised with a string parameter `raise x("hello");` and caught with a handler expecting a string `catch x(p:string).print p;`. The type checker can easily check that in both cases a string is being used.

The problem is to guarantee that the x we catch is the same as the x we raised. We need some identification when we raise an exception which the handler can look at to decide which exception this is. If we can declare local exceptions with scope in a similar way to local identifiers we may have two exceptions both called x with different argument signatures in separate parts of the same program. We could pass a procedure which raises one x but does not catch it into the scope of a handler for the other x. Since the parameters have different signatures the handler must treat this as a different exception even though it has the same name. This problem can be avoided if the compiler generates a unique identification for each exception. Unfortunately this will not work properly either, if we have exceptions with polymorphic parameters.

```
let p ==
proc(t: type end; b: boolean; x:t; f: proc(t)t)t
   begin
   let ex == exception(t); { Declare exception x }
   if b then raise ex(x)   { Raise the exception }
```

```
   else
      begin
      f(x) { Call f, which may raise an exception }
      catch ex(y: t). y { Catch ex, returning the parameter. }
      end
   end;
let q ==
proc(i: integer)integer
   begin
   let z ==
      p(string,
        true, { Raise an exception with a string parameter. }
        "hello", { A string }
        proc(s: string)string (s) { Not used }
       );
   i+1
   end;
let result: integer ==
   p(integer,
     false, { Catch an exception }
     1,     { Unused }
     q      { proc(integer)integer but raising ex with a string }
     );
```

p is a procedure which either raises an exception or calls a procedure depending on the value of b. It is polymorphic since it is parameterised on the type, t, used in the signatures of x and f and in the result signature. Within q t is string and so the exception ex is raised with a string argument. In the second call to p the type is integer. This time p is catching the exception, but expects an integer and returns it as an integer. If the exception is the same in both cases so that the exception raised in one call to p can be caught in a different call the type system can be broken. The only way to keep the type system secure is to say that different invocations of p generate different exceptions. This requires the exception declaration to create a value dynamically which will be the exception identification for ex in that particular scope. Within the scope all references to ex will find the same identification, but if the exception declaration is executed again a different identification will be created, and the exceptions will not match.

2 Proposal

We introduce a new class of signature, the *exception* to rank alongside types, procedures and simple values. An exception has an argument list exactly like a

procedure, but no result signature. The argument list can include implied arguments. A missing argument list is treated as though a single argument of type *void* had been given.

Syntax		
<signature>	::=	**exception** <argument list> | **exception**

```
rangeerror: exception
failure: exception(string)
print_ex: exception[t: type (t) print: proc(t) end](x:t)
```

An exception is a value and is created by the *exception constructor.*

Syntax		
<expression>	::=	**exception** <argument list> | **exception**

```
let rangeerror == exception
let failure == exception(string)
let print_ex == exception[t: type (t) print: proc(t) end](x:t)
```

It will normally only be used in a declaration but it is possible to pass it as a parameter or to return it from an if-statement or a procedure. It can also be a component of a type.

An exception is raised with the *raise expression.* The arguments supplied in the expression must match the signature of the exception. The rules are the same as for a procedure.

Syntax		
<expression>	::=	**raise** <identifier> <expression> | **raise** <identifier>

```
raise rangeerror
raise failure "It failed"
raise print_ex 2
```

It would be possible to have an arbitrary expression in place of the identifier but it seems so unlikely that, for the moment at any rate we will keep this restriction. The only case that is at all likely is a selection from a type. Having this restriction also allows us to have a name to print if we catch an exception in an **else** catch phrase.

An exception is caught with the *catch phrase* at the end of a compound expression.

Syntax		
<handler>	::=	**catch** <catch phrase>; ... ; <catch phrase>
<catch phrase>	::=	<identifier> <parameter list> . <expression> |
		<identifier> . <expression> |
		else <parameter list> . <expression> |
		else . <expression>

```
catch rangeerror           . print "Overflow";
      failure (x: string) . print("Failed with " + x);
      print_ex [t: type (t) print: proc(t) end](x:t)
  . (print "Exception "; print x);
      else (x: string)    . (print "Unknown "; print x)
```

The general form of the catch phrase consists of a named exception followed by a parameter list and an expression to execute. As with the raise expression this could be generalised to allow arbitrary exception-returning expressions. When an exception is raised the exception value, *not* the actual name, is compared with the values in each of the catch phrases until a match is found. The actual parameters given in the raise expression are used as the values in the parameter list, if there is one, and the corresponding expression is evaluated. The signature of the parameter list must match the signature of the exception.

The **else** form of the catch phrase catches all exceptions. The parameter list if present must have a single string parameter. If it catches an exception that parameter will be set to the name of the exception used in the *raise expression.*

Structure-Editing in Poly and ML

David C.J. Matthews

This note is a suggestion for the way to proceed with structure-editing in Poly and/or ML. It should be relevant to both the Programming Logics project and to other work in Poly and ML. N.B. throughout this note I have used structure to mean a data structure, typically a tree. When the word is used in the sense of a module in ML this is made clear in each case.

1 Type System

Existing structure-editors, such as the Mentor system from INRIA, have a type-system for the structures to ensure some consistency. One wants to be able to declare, that when editing a parse-tree for example, a tree denoting a declaration cannot be moved to a place in the tree where an expression is expected. The editor can inform the user that a particular operation would break the type-rules in the same way that a compiler will refuse to compile a statement in a programming language which is type-incorrect.

Operations performed directly by the user on the tree require dynamic type-checking, that is each operation must be type-checked when it is invoked. It is not possible for a programmer to decide in advance what sequence of operations a particular user is going to want, so it is impossible to type-check it beforehand.

There is however a place for static type-checking in such a system. It is often convenient to be able to write sequences of editing commands together and treat them as subroutines to be invoked from the console. Since we have a dynamic type-checking system for the primitive commands we could use that for the subroutines by simply checking each primitive command as it is applied. It is much better, though, to check the subroutine when it is created to ensure that it is type-correct. It should be possible, with an appropriate type-system, to check that the application was valid in a particular context, i.e. that its arguments are correct, and then the internals need not be checked on each application.

The structures we create and manipulate with the editor are going to be used by other programs. If we are editing the structure of some proof we want to be able

to run the proof-checker on the resulting structure. The data structure used by the proof-checker will probably be an ML datatype. It would be sensible if the structure editor, rather than operating on a parallel data structure, and having the ML program translate it into the proof-checker's structure, could use the same structure.

This has several advantages. The immediate saving is in the cost of translating the data structures. It also provides us with a type-system for the structure. Transformations are allowed only if they conform to the ML type declarations. Operations on the tree are performed by ML functions, usually executed dynamically from the console, but it is possible to construct more complicated functions by composing the primitives. Because these would be written in ML the normal type-checking rules would be applied and in addition they could perform arbitrary computations.

Editing ML datatypes is relatively easy because the datatype definition gives a list of the constructors and their types. So, given a datatype value and its type definition the editor can break the structure down and display it.

There are a few disadvantages. The most obvious is that ML datatypes cannot in general be destructively updated. There are many reasons for not allowing the editor to have a privilege forbidden to other programs, so instead we must settle for allowing only copying of structures with replacement. This should not be too expensive since the editor can keep track of which parts of a tree have been modified and which have not.

2 User Display Functions

The editor can use the datatype definitions to print out the structure, but this will only be in a general fashion. It would be much better if the user could write a function to display a particular structure which the editor could then invoke. Unlike Poly, ML has no way of attaching functions directly to datatypes so the only way of doing this would be either by using a name such as `T_display` to display a value of a type called `T`, or having a structure (in the sense of a module in ML) `T` containing a function `display`. As well as the value to be displayed the function would have to be given the size of the window on the screen in which the value is to be displayed. In this way it could calculate how much of the structure to show.

User display functions also present problems with "picking" values from the screen. We want the ability to select a node by pointing at something with the mouse. This means translating screen co-ordinates into a node, requiring an inverse of the display function. Since the user's display function laid out the text on the screen, only a similar function can interpret some screen co-ordinates as a node

in the structure. Apart from the problem of actually finding the node from the co-ordinates there is also the problem of the type of the value returned.

```
datatype expression =
  num of int | ident of string | let_in of declaration * expression
and declaration = Declaration of string * expression
```

This is the parse tree for a simple language with numbers and identifier declarations. It contains values of various types any of which should be able to be selected by the "pick" function. If we selected a number we should be able to get that number out of the structure. We could create a new union type containing all the types in the structure and declare the "pick" function as returning a value of that type. Alternatively we could introduce the idea of a universal type similar to dynamic in Amber, and coerce the value to that type. While the idea of introducing dynamic type-checking in ML seems a step backwards it is really the only solution to this problem.

We introduce two infinitely overloaded functions, `dynamic`, which can be applied only to values of types where all the type constructors have global scope and and `coerce`, whose result must have been unified to such a type. `coerce` raises an exception if the value cannot be unified to the result type.

```
dynamic: 'a -> dynamic
coerce: dynamic -> 'a
exception coerce: unit
```

Dynamic values contain a value together with its type and are the objects that the compiler uses when top-level declarations are made. Picking a value on the screen would return a dynamic value. It could probably also return a function which would accept a value of the appropriate type and rebuild the structure with it. The editor can then ensure that a structure is only rebuilt with the appropriate values.

3 Structure Editing in Poly

In Poly the implementation of a type is hidden and the only information available is the signature of the operations. A type defined as a union is initially created with injection, projection and tag-test operations, but it is easy to redefine these, add new operations or hide them. Unlike a datatype in ML it is not possible to examine the type to decide how to take a value apart. However in Poly it is relatively easy to associate a function with a type so we could insist that only structures with the

appropriate functions can be edited, and not provide any default display behaviour based on the type definition. As with ML we will have to introduce dynamic type-checking. Rather than have `dynamic` and `coerce` as functions these would be reserved words with a syntax similar to function applications. This allows the restriction that only values involving types with global scopes to be enforced. The other reason is that because type checking is not done by unification the signature that `coerce` is to return must also be passed as an argument.

4 Operation of the Editor

The editors working on ML or on Poly will be fairly similar apart from the way they construct functions from a type when given a value to edit, so for the moment we will draw the examples from ML. We can think of the editor as a front end to the ML compiler. It generates strings of ML which are passed through the compiler and executed. While in practice it would be more tightly bound into the compiler so as to speed up the process, it is useful to think of the operation in this way.

A structure is edited by passing a value of type `dynamic` and a window to the editor. Normally this will be done by giving the name of a structure stored in the global name space and creating a new top-level window. The editor examines the type of the value it has been given and then generates a call to the appropriate display function, passing it the value and the window.

```
T_display(ROOT, window);
```

The display function recurses down the structure displaying it in some suitable form. Since it is given the area of the window it has to work in it can choose to display as much as will fit and elide the remainder. If the mouse button is clicked in this window the pick function for this type is called. This either returns the node selected or raises an exception if the co-ordinates do not lie in an identifiable part of a node.

```
val PICK =
   let val picked = T_pick(ROOT, window, mouse_x, mouse_y)
   in coerce(picked): type_of(picked)
   end
```

The above would not be valid ML even with the addition of dynamic types but it illustrates the kind of code that would be generated. The point is that the result held in `PICK` is a value of a particular type depending on the type of the value

selected, not a dynamically typed value. If a number is selected the result will be that `PICK` is declared as the value of the number and type `int`. This is rather like "`it`" in ML which holds the value of the last expression evaluated. Its type depends on the type of the result of the expression.

There are two reasons for selecting a node, we may want the node itself or we may want to replace the node with a value from somewhere else. Selecting a node can be used as a way of zooming in on part of a structure because having selected a node we can then display the structure in the same or another window using the selected node as the root.

Replacing a node cannot be done directly, instead we have to generate a new structure by copying the old one with the replacements. Regeneration must in general be done by a user function because there may be other constraints on the data as well as type-consistency which must be preserved. If we are editing an ordered list of integers we cannot replace a particular node by an arbitrary value. We have an analogue of the "pick" function, called "poke" which rebuilds the structure with a particular node replaced by a given value. An alternative would be to have the "pick" function return a function to rebuild the structure as well as returning the current value. Replacing the currently selected node involves applying this function to the new value.

```
val ROOT =
  T_poke(ROOT, dynamic NEW_VALUE, window, mouse_x, mouse_y);
```

The new value is passed in as a dynamic because the type required depends on the position in the structure where the new value is to be placed. The result is a new root for the structure.

5 Conclusions

1. We have to introduce dynamic types into ML and Poly.
2. For each structure we wish to edit we must provide `display`, `pick` and `poke` functions. These are responsible for the layout of the structure and any consistency checks on the data. They will typically be written in terms of `display`, `pick` and `poke` functions of other types.
3. The editor will take care of overall screen management. It will have to have some information about the data structures used to represent ML and Poly types in order to be able to generate appropriate function calls. It might be possible to abstract some of this out to make the editor more independent of the type system but I expect there will still have to be some dependencies on the type system.

Static and Dynamic Type-Checking

David C.J. Matthews

Abstract

The purpose of a type checker is to prevent an incorrect operation from being performed. A static type checker does this by stopping the compiler from generating a program with type errors, a dynamic type checker halts the program as it is about to make a type error. It is clearly useless to have a dynamic type checking system for a program which is to be produced, distributed and used by anyone other than the original authors since any type errors that occur would be meaningless to the user of the program.

On the other hand, where a user is guiding a program through some data, a dynamic type-checking system is reasonable. Examples are browsing through a database or structure-editing. Here type-errors have meaning to the user.

The ideal language would be basically statically type-checked but would allow dynamic type-checking when necessary. While this is possible with certain type systems there are others for which it is difficult. The implementation of dynamic type checking in various type systems is considered.

1 Type-Checking

Type checking is an effective way of reducing programming errors. Its function is to identify those values to which an operation can be "sensibly" applied. The definition of "sensible" depends on the type system but usually operations like adding together two functions are not regarded as sensible while adding two numbers is.

1.1 Static and Dynamic Checking

One way of doing the type checking is to tag each value with a few bits which describe its type. Each operation checks the tag bits and gives some sort of failure if the values have the wrong type. *Dynamic* type checking will prevent some of the more obscure errors but has the disadvantage that the failure is only generated when the program is run.

A better method involves placing some restrictions on the programs that can be written so that the compiler can decide *statically* whether a program could possibly generate type failures when it is run. If the program can be shown to be type-correct there is no need for the tags and we know before the program is ever run that type errors will not occur.

1.2 Binding

Related to this is the question of binding. Declarations bind names to values and so have types. When an identifier is looked up the value with its type is returned. If there are several identifiers with the same name there must be rules for deciding which one is meant in a particular context.

One of the restrictions for static type checking to be possible is that the compiler must know the type of all the identifiers in the program. This requires *static binding* to identifiers, that is the identifiers are matched up with their declarations when the program is compiled and does not depend on the execution paths.

It is possible to have *dynamic binding* where an identifier is looked up when the program is run, but static type checking is possible only if all the identifiers with the same name have the same type. General dynamic binding requires dynamic type checking.

2 Static Type-Checking

Testing a program to try and find errors is difficult, and can never guarantee correctness. The ideal programming language would be one which imposed no restrictions but where the compiler could decide whether the program was correct. Unfortunately that is impossible and so we must accept some restrictions and even then we only have a limited form of correctness. However type-correctness is sufficiently useful that paying the penalty in terms of accepting some restrictions is reasonable. Recent developments in the design of type systems, particularly polymorphism[3], have extended the range of static type checking into areas where traditionally dynamic type checking was thought necessary.

2.1 Type Equivalence

At the lowest level a type must describe the structure of its values in terms of type constructors such as records and unions and the primitive types of the language, in order that the primitive operations can be type checked. In one form of type checking, if a type can be given a name it is treated as an abbreviation for the structure and two values are treated as the same type if they describe the same structure. This is *structural type equivalence* used for example in Algol 68[9].

An alternative is to define that two values have the same type only if they have the same type name. If the type names are different the types are incompatible even if the structures of the types are the same. *Name equivalence* does not mean that the structure of the type is not visible, only that it is not used for type equivalence. *Abstract types* are a variation of name equivalence where the association between a type name, the *abstraction* and its structure, the *implementation* is only visible within the abstract type definition[4]. Outside that it has no structure and name equivalence is used. When types can be returned as a result of functions name equivalence or a variation of it is needed to ensure that type checking is decidable[2][6].

3 Dynamic Binding and Type-Checking

Static type checking is useful when a program is being produced which is to be executed later. If the program is to be executed immediately, and particularly if it just consists of a single command, there is really very little difference between static and dynamic type checking. Command line interpreters, or "shells" are an example. The command

```
edit afile
```

typed to a command interpreter would probably involve a search for the files `edit` and `afile` and checks that `edit` was an executable file and `afile` was a text file. The search and type checking for `afile` might well be done from within the `edit` program. There is no advantage in treating type checking separately from execution.

However, if several commands are put together in a command script it starts to look more like a writing a program. Because the individual commands are dynamically bound and type checked it is not possible to statically type check the completed script even though it resembles a programming language procedure. Apart from command interpreters the Mentor programming environment[7] is another example where structure editing commands in the Mentol language can be put together into procedures.

Apart from the fact that there is no advantage in statically checking a command which is to be executed immediately, there are other reasons why dynamic type checking is used. Programs often create file names, for instance by appending standard suffixes onto a name to make a set of related file names. Since the files are dynamically bound they must be dynamically type checked.

Dynamic binding may not only be by name but by other mechanisms as well. In a structure editor a user may select an item by pointing to it with a mouse. Different parts of the structure will have different types so that changes to the structure are constrained, but the function that returns a selected value must be able to return a value of any type. Dynamic type checking must be used if this

value is to be copied somewhere else in the structure.

4 Combining the Two

Static type checking is needed for programs which are to be executed in the future, but dynamic type checking is needed for interactive operations. If we have a system where both of these activities can occur we really need both mechanisms.

The obvious way to do this is to take a static type system and add some additional syntax and a new type *dynamic*.

dynamic x

constructs a value of type *dynamic* by packaging up the value with information describing its type. The inverse operation

coerce d **to** t

checks that d is a dynamic value with the type information appropriate to the type t and returns the original or raises an exception. The syntax is taken from Amber[1]. Dynamic binding can be done by returning values of *dynamic* type and then coercing them to the appropriate type.

A dynamic value contains a value and a representation of the type. The type representation must contain enough information for the **coerce** operation to do the same kind of checking at run-time as the compiler would do at compile-time. We do not want the dynamic type mechanism to subvert the static type system. This may be more complicated than it appears. The rules for static type equivalence which are applied at compile-time may not be reproducible at run-time. To see how the static type system influences the dynamic typing some static type systems will be examined, both from languages which have dynamic types and those that do not.

4.1 Structural Equivalence

The simplest type systems for this purpose are those such as Amber that have a fixed number of primitive types and use structural equality between types. Each primitive type can be assigned a unique identifier and data structures used to describe the structured types. Because names for types are just synonyms for the structure we can always use a representation of the structure for the type representation. Although the name may be declared locally the structure representation is valid anywhere so dynamic type checking is safe. Type inheritance in Amber does not have any serious effect on this.

4.2 Polymorphism

If the language allows polymorphic operations, as in ML, dynamic type checking has to be arranged more carefully. Consider the following two functions.

fun *get_dynamic* d = **coerce** d **to** α;
fun *make_dynamic* x = **dynamic** x;

get_dynamic takes a dynamic value and coerces it to the type variable α. If the dynamic type matching rules follow the static type rules these should match for any type since unification of a type variable with any type would succeed. Clearly this would allow the type system to be broken because we could make a dynamic value out of, say, an integer value, pass it into *get_dynamic* and treat the result as a string.

make_dynamic will take any value and make a dynamic value from it. This again could break the type system. A solution to both of these problems is simply to forbid polymorphic types in **coerce** or **dynamic** operations.

4.3 Abstract Types

If the static type system allows the user to create abstract types we have to produce a unique identifier for each abstract type and use those in type representations.

A dynamic value created from the abstract type will contain different type information to a dynamic value created from the representation. This may be difficult if the dynamic value is created inside the abstract type package since there the distinction between the values of the abstract type and the implementation type is blurred.

4.4 Parameterised Types

Parameterised abstract types create another problem. If we have a type which can be parameterised by other types or values we need to ensure that any dynamic values created from values of the result type or the argument types have the correct type representation. If the parameterisation simply involves macro-expansion this is relatively easy but if it is done at run-time the type representations will have to be passed as run-time values. The type identifier for the resultant type may have to be created dynamically when the parameterisation is done.

For example, we may define a type *tree* which is a binary tree parameterised by the type of the leaves. Inside the type definition we write an operation to walk over the tree in response to commands from the user and return either a leaf or a piece of tree as a dynamic type.

```
abstype α tree = Leaf of α | Tree of α tree * α tree
with
  fun move "value" (Leaf l) = dynamic l (* Return the leaf *)
    | move "value" (Tree t) = dynamic t (* Return the tree *)
    | move "left" (Tree (l, _)) =
          move (nextcommand()) l (* Move left *)
    | move "right" (Tree (_, r)) =
          move (nextcommand()) r (* Move right *)
    | move _ _ = raise bad_command (* Anything else *)
end
```

In this example Standard ML[8] has been used with the addition of the operation to create dynamic types. The **dynamic** operations have been applied to polytypes

which would allow the type system to be broken, so clearly this cannot be allowed as it stands. In any case it is difficult to see how it would achieve what is wanted, which is for leaves of *int* trees to be returned as dynamic values coercible to values of type *int*.

However if we treat the parameterised type more like a function so that the parameterised type is always used in its parameterised form we can safely allow dynamically typed values to be created and probably get the required behaviour. In Standard ML this could be done using a parameterised module, called in ML a *functor*[5].

```
functor Tree(Elem: sig type t end) =
  struct
  datatype tree = Leaf of Elem.t | Tree of tree * tree;
  fun move = ... (* As before *)
  end
```

The type *tree* is only available when the functor *Tree* has been applied to a module, in ML a *structure*, containing a type. Other rules in ML ensure that this is a monotype.

```
structure IntTree = Tree(struct type t = int end);
```

This creates a tree whose leaves are integers. In order to get the effect we want the type representation for *int* must have been passed into the functor so that the dynamically typed values returned from leaves are recognisably integers. The tree type *IntTree.tree* itself is a new atomic type so the representation for the type can be created dynamically when the functor is applied.

In CLU, which has parameterised clusters and dynamic types, this is rather more difficult. A cluster parameterised by the same parameter values denotes the same type wherever it appears in a program. Since the type may appear in different segments the CLU linker must examine all the types, construct unique identifiers for each different type, and then pass the identifier to be used for the result of each parameterised type into the type as an additional argument.

4.5 Types as Values

In languages such as Russell and Poly types can be treated as first class values. A type and operations associated with it are packaged together and treated as a run-time value. To ensure decidability name equivalence has to be used. An expression such as

let *atype* == **if** ... **then** *type1* **else** *type2;*

declares *atype* to be a type which is not the same as either *type1* or *type2*, since it is in general not decidable which is actually being returned.

This causes problems if we try to use the dynamic type scheme suggested above for ML. Suppose *type1* and *type2* are different implementations of trees as in the ML example. They both have *move* functions which return dynamically typed values. If we make a tree of type *type1* we expect the dynamically typed values to be coercible to *type1* but not to *type2* or to *atype*. Similarly values from *atype* should not be compatible with either *type1* or *type2*. Unfortunately if the type representation is put into the dynamic values inside the abstract type declaration the dynamic values returned from *atype* trees will be either *type1* or *type2* depending on the actual type returned by the **if**. There seems to be no way to avoid the dynamic type checking behaving differently to the static type checking.

5 Conclusions

Certain applications require dynamic type checking in an otherwise statically typed language. For some type systems this is relatively easy to arrange, but others require considerable thought if the security of the static type system is not to be undermined.

References

[1] Cardelli L. *Amber* AT&T Bell Labs Technical Report 1984.

[2] Demers A. and Donahue J. *Revised Report on Russell* TR79-389, Department of Computer Science, Cornell University, 1979.

[3] Gordon M. et al. *A Metalanguage for Interactive Proof in LCF* Fifth Annual Symposium on Principles of Programming Languages, Tucson 1978.

[4] Liskov B. et al. *CLU Reference Manual* Springer-Verlag, Berlin 1981.

[5] MacQueen D.B. *Modules for Standard ML* AT&T Bell Labs Technical Report 1985.

[6] Matthews D.C.J. *Poly Manual* SIGPLAN Notices. Vol.20 No.9 Sept. 1985.

[7] Mélèse B. et al. *The Mentor-V5 Documentation* Technical Report 43, INRIA 1985.

[8] Milner R. *A Proposal for Standard ML* in "Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming", Austin, Texas 1984.

[9] van Wijngaarden A. et al. *Revised Report on the Algorithmic Language Algol68.* Springer-Verlag, Berlin 1976.

A Persistent Storage System for Poly and ML

David C.J. Matthews

July 6, 1988

Abstract

The conventional strategy for implementing interactive languages has been based on the use of a "workspace" or "core-image" which is read in at the start of a session and written out at the end. While this is satisfactory for small systems it is inefficient for large programs. This report describes how an idea originally invented to simplify database programming, the *persistent store*, was adapted to support program development in an interactive language.

Poly and ML are both semi-functional languages in the sense that they allow functions as first-class objects but they have variables (references) and use call-by-value semantics. Implementing such languages in a persistent store poses some problems but also allows optimisations which would not be possible if their type-systems did not apply certain constraints.

The basic system is designed for single-users but the problems of sharing data between users is discussed and an experimental system for allowing this is described.

1 Background

There has long been a division in programming languages between those which are used in essentially a "batch" mode, where a source file is compiled into an object file, and the interactive languages, where the source text is compiled as it is typed in. The differences are in a sense more to do with the implementations than with the languages themselves and even in the interactive case there is usually a command to compile source text from a file. On the whole though languages tend to lend themselves better to one implementation or the other.

Apart from the way in which programs are constructed there is another difference between interactive and non-interactive languages. Non-interactive languages are compiled into separate programs and any communication between them must be through the operating system. This usually means through a text file or something similar. In contrast in an interactive session functions, or the equivalent, are applied to generalised data and generate data. The data may often be structured, such as lists or trees. Within a session this is easy to arrange, but we would often like to be able to suspend a session and come back later. This means that structured data have to be converted into a file that can be held in the filing system. The non-interactive system, which works on files anyway, does not have the same problem.

The conventional solution is simply to write out the contents of the memory at the end of the session and then read it back in at the start of the next[Fal67]. This "workspace" or "core-image" idea allows a user to develop a system over a number of sessions. It has the advantage that the structuring and naming used is appropriate for the language rather than being imposed by the operating system. The disadvantage is that the facilities provided by the operating system for sharing between users is not available. The workspace has to be treated as a single entity and it is difficult to isolate a useful function from the workspace and make it available to other users, though[Fal67] describes how a variable or function could be copied into the active workspace from another. In addition, since the whole workspace has to be read in to store there is the problem that it may grow too big to fit into the available memory. Solutions to this problem[Bob67][Ing78] have tended to require changes to the operating system rather than a system which would run on top of an existing system without modification.

2 Persistence

The idea of using persistent storage as a convenient way of implementing databases was proposed and implemented by M.P. Atkinson[Atk83]. He noted that programs operating on databases often have two representations of data, one used internally

by the program while it is running, and a different representation for the same data when held in the database or filing system. The program was forced to spend a considerable time translating between the two representations, because the types provided by the database for persistent data were different from those provided by the programming language for transient data. His solution was to suggest that persistence of data should be a property which could be possessed by a value independently of any other property. In particular whether an object could persist was not associated with its type. The programmer could then choose a representation for the data purely on the basis of the algorithm and not in order to satisfy the requirements of a database system.

The language PS-Algol[Atk81] was designed to test these ideas. A PS-Algol program can open and operate on data in a database, modify it and add new data to it. There are functions to open and close the database and to *commit*, or write back changes. These same operations would be present in any database system. The difference with PS-Algol is that there are no other operations to read or write data. Instead objects are read in as they are required. The programmer can use any data structures appropriate to the task and the storage system will ensure that objects are brought into store. In this way persistent data are treated in exactly the same way as transient data and can, for example, be combined together in a single structure. Changes are made by the normal assignment operation but are only recorded in the database when the explicit *commit* is called. The action of *commit* is to preserve in the database every object which could be found by following pointers from a number of distinguished roots. Since these roots are the only way in which a program can get access to the data this rule ensures that all the useful data are preserved.

A persistent store can be thought of as a cross between a virtual memory system and a database. Objects in a database can refer to other objects and any object can only be reached by following these references from a few well-defined roots. Transfers to and from the database are usually made by calls to special procedures. In a virtual memory system pages are transfered to and from backing store without any explicit requests from the user. They are regarded simply as unstructured store which, if it can be retained in the filing system at all, must be retained as a whole. A persistent store combines these ideas by having automatic transfers but retaining only reachable objects in the database.

Because objects are read from the database transparently the system behaves very much like one where the whole workspace is read in and written out at the end. Opening a database is similar to reading in the workspace and *commit* to writing it out again. The difference is that the cost of reading a persistent database is dependent on the amount of data used and not the overall size of the database.

3 Poly and Standard ML

Poly [Mat85] and Standard ML [Mil84] are general purpose programming language supporting polymorphic operations. They are both statically type-checked and statically scoped and treat closures as first-class objects. Their type systems are different but the underlying abstract machines are sufficiently similar for a common implementation to be used.

They are used interactively and in the Poly/ML system the two languages together comprise a single system. The original implementation of Poly, and other implementations of ML, have used the workspace idea to preserve data from one session to the next. When the size of the workspace became large an alternative was needed and a persistence storage system, based on that for PS-Algol, was designed.

The initial design of the persistent store for Poly and ML was similar to the PS-Algol work. However as the design progressed it became clear that there were properties of Poly and ML and the way they are used, that would affect the design.

It was regarded as important that the system should be transparent to the user as far as possible. The user should not have to think in terms of a database but in terms of the programming language and his own data structures. The persistent store in PS-Algol went most of the way towards this by making transfers of data from the database into store transparent. As far as the user of Poly or ML is concerned the system is very much like using a core image which gets read in at the start of the session and can be written out when required. The main difference is that the initial prompt appears almost immediately but there is a delay when the first command is executed as the compiler is brought into store. At any time the user can call a **commit** function which will write changes back to the disc. Changes are also normally written back at the end of the session.

Both languages are strongly type-checked and make clear distinctions between values which can be updated, *variables* or *references*, and those which cannot. These features are exploited in two ways. Since the type system will prevent addresses being used except as references to objects the system can operate on the addresses without the user being aware. This would not be possible in a language which allowed the user to extract the address of a word inside an object, for example. Distinguishing updatable, or *mutable,* objects makes it is possible to mark them when they are created so that the system can keep them separate from the non-updatable, or *immutable,* objects. In practice the vast majority of objects are immutable so allowing some optimisations which would not otherwise be possible.

The Poly and ML compilers are written in Poly and are part of the system. All the compiled code they generate and all the other data structures are also within

the system. Unlike in PS-Algol, where programs from outside can operate on the data and so there have to be ways into the data from outside, in Poly or ML the only operation needed is to start running the *read-eval-print* loop of either Poly or ML. The root of the database is therefore just a procedure which is called from the system when the session starts.

4 Implementation of Persistence

Reading in a single core-image, operating on it and writing it out at the end is fairly simple. The addresses in the image may have to be relocated but this need just involve adding a fixed offset.[1] The relocation is done all at once for the whole system. After relocation the addresses are the normal memory addresses for the machine so compiled code in the memory can operate on them. If however, we go to a form of paging where some objects may be in memory while others are on disc things become more complicated. If an object may or may not be in memory its address cannot be a simple memory address. An address must give, in some form, the location on disc where the object is to be found and it will have to be translated into the real address after the object has been loaded. In a virtual memory system this translation is done each time an address is used, and because it is supported by hardware or micro-code that is acceptably fast. However if we are using software to do the translation, calling a subroutine every time we used an address would be far too slow.

The solution used in PS-Algol and adopted in this system was to allow the two forms of the address, the disc or *persistent* address and the address of the object in memory, the *local* address, to co-exist in the memory. In principle, local addresses are used for objects actually in the memory and persistent addresses for those on the disc. Whenever an object is read into memory all the persistent references to it are changed into local addresses. In practice this would require a complete scan of the memory whenever an object was read in to find all the references, so instead we only overwrite a persistent address when it is actually used.

By choosing the local address to be the actual memory address of the machine we can run ordinary machine code programs. It is still necessary to look at an address before it is used to decide whether it must be translated, and call the translator if not. This could be reduced to a few instructions but if we wish to run machine code they would have to be compiled in every time an address was used to refer to an object, in case it was a persistent address. If we choose values for the persistent addresses which are invalid as memory addresses, and we can usually find some part of the address space which is illegal, we can make use of the machine to distinguish addresses for us. We compile in a normal instruction

[1] We assume that it is possible to distinguish the addresses and data in an object.

to use the address as though it were an ordinary local address. If the code is run on a local address it will run normally, but using a persistent address will result in an illegal address fault being generated by the hardware and operating system. If this fault can be passed back to the persistent storage system we may be able to overwrite the persistent address with its local equivalent. The instruction which made the trap can be restarted and we will be able to continue executing as though a local address had been used all along.

There are two trade-offs made here. The first is that having the two addresses coexist means that local addresses will have to be translated back into their persistent form when objects are written out. The second is that taking and catching an address fault is likely to be expensive since it involves calling the operating system. If a particular object is used only once then the cost of loading it and later writing it out is going to be expensive. However in the kind of applications for which this system is used most objects are used repeatedly once they have been read in and the average cost is much lower. A more serious problem is that we may have several copies of an address. If we use a persistent address it will be overwritten by its local form, but other copies will not be, and we will get faults if we try to use those. There are various strategies which are used to overcome this.

5 Constraints on the Code

There are at present implementations of the system for the VAX and the Sun (MC68020), and there is a portable version using an interpreted code where the persistent addresses are detected by the interpreter. The persistent storage system is the same in each implementation apart from a small section involved in decoding an instruction when a trap has occurred and in restarting the instruction. This is obviously machine dependent and will not be described in detail.

The system relies on being able to restart instructions after a persistent address has been overwritten by its local equivalent. An instruction may be part way through executing when the invalid address is detected, but in principle this should be no problem. The hardware of a machine which supports virtual memory must be able to restart any instruction after a page fault. On the VAX, where the internal state is backed-up by the machine to a point where the instruction can simply be re-executed this is true, but on the MC68020 a trap results in internal state being saved. This can be reloaded after a page fault under control of the operating system but is not available to a user process. This means that a general instruction cannot be restarted after a persistent store fault. However the code which is operating on the persistent store is generated by the Poly code-generator and it can be written to avoid instructions which would cause problems. Where this is not possible such instructions can be preceded by another instruction whose

only function is to cause a persistent store trap on an address so that the next instruction will always have a local address to deal with. Adding these instructions means that code is slightly larger than it would be if it were working entirely on normal memory addresses but much less than if code had to be included explicitly to distinguish persistent addresses.

The other constraint on the code which may form part of the system is that garbage-collection may cause addresses to change. As well as being called when an object is created on the heap it may also be called when a persistent store fault occurs and space is needed for the object being read in. This may affect the code which can be generated.

6 Address Translation

A core-image is usually a contiguous file in which addresses correspond more-or-less to an offset in the file. We could use that for a persistent store and a persistent address would then be the offset of an object from the start of the file. When an object was written back to disc it would be written to the original place it was read from. Unfortunately if the machine or the program crashed during the writing process it might leave the file in a state where some objects had been written back and others had not. To avoid this problem we always write back blocks to areas of the disc which were not previously in use and never overwrite something useful. Since objects are written out to a different place from where they were read we need a map to give the current location of an object. The scheme is basically that suggested by Challis[Cha78].

We also need a map between persistent and local addresses so that we when we write an object out again we can translate the addresses in it back to their persistent form. It will also be used to translate other persistent addresses to an object which is already in store where there are several references to it. These maps both translate persistent addresses to either disc locations or local memory locations and for convenience they are combined into a single *address map.*

When a persistent address causes a fault we check in the map to see whether the object is already in store and if it is not we read it in. This will involve the operating system reading it from disc which it will almost certainly do by reading a disc block and extracting the bytes required. Since most objects are small there will be several in a disc block and it may be read several times to extract each object. Provided there is some locality of reference it may be better to extract all the objects from the block once the block has been read in. There is a problem with objects that straddle block boundaries so a better solution is to arrange that a set of objects always starts on a block boundary and waste a little space at the end of a block. If an object is larger than the block size we have to use more than

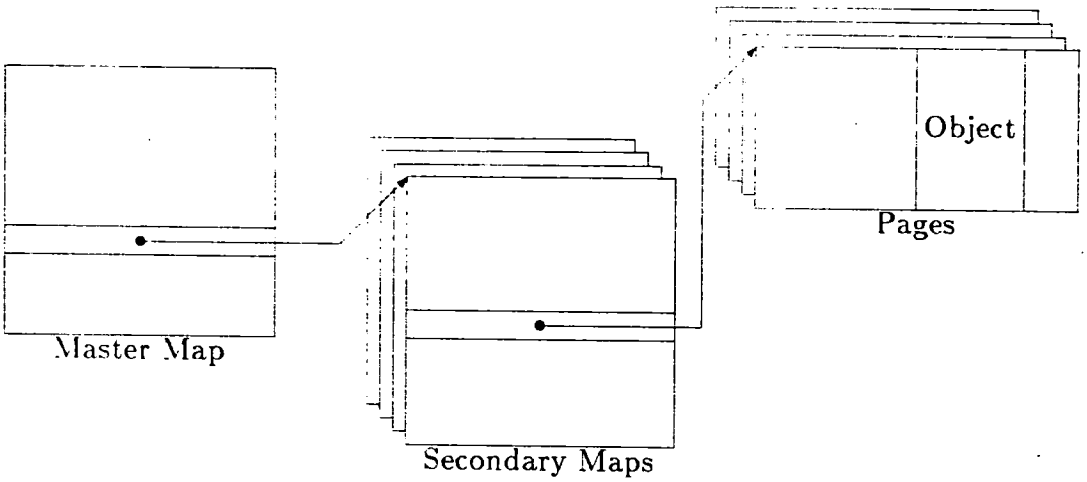

Figure 1: Map Structure

one block for it. Since it will probably not be an exact multiple of the block size there will be spare space in the last block but this is left unused[2].

In practice we may not know the size of disc blocks on a particular machine so instead we assume a value for the *page* size which we hope will be a multiple of the block size. It is convenient to use a power of two so that the a persistent address can easily be separated into a page number and an offset. Grouping objects into pages also simplifies the address map. Since objects are always read in as a page the map need only give the address in memory or on disc of the page and not of each individual object.

Even though there is only one entry in the address map for each page it would still be fairly large if it were a single vector. Instead it is split into two levels with a master map giving the location of a set of secondary maps which in turn give the addresses of the pages 1. Translating a persistent address into a local address involves looking at the master map to find the secondary map, reading that into memory if it is not there and then reading the individual page if that is not in memory.

Using a two-level map like this has another advantage. When we come to write the memory back to the disc we first write the pages to new locations on the disc and then write back the secondary maps which give these new locations. These are also written to new locations on the disc and these are recorded in a new version of the master map. It is only when all of these are safely back on the disc that we write out the new master map. By following this procedure we have the database in a consistent state at all times. If a crash happens at any point before the master map is written, the database is in the state it was in before we started to write to

[2]This is because if we filled it up with small objects we might want to read in one of them before we read the large object, and end up with part of the large object read in.

it, because reading will always involve first finding the master map and then using that to find the secondary maps and the pages. Starting with the old master map will find the database in its old state, starting with the new master map will find it in the new state.

7 Optimisation

Unlike a conventional virtual memory system, the cost of dealing with a persistent address trap is considerable even if the object is already in store. The persistent store handler overwrites the address which caused the fault so there will not be another fault if that address is used again. That would suggest that there will be at most one fault for each copy of each address in all the objects read from the store. Unfortunately this does not work out in practice. Typically an address is loaded from store into a register and the register is used to find an object. The persistent store system will overwrite the copy in the register but the value in store will be unchanged. Repeating the process of loading and indirecting will cause another fault, and the only way to prevent it is to find and update the copy in store. The problem with that is that the cost of searching for it may be more than the cost of a few extra store faults.

Various methods are used to reduce the number of persistent store faults.

- When a new page is read in all the addresses which refer to objects already in store or in that page are updated.
- Clustering objects which refer to each other in pages will reduce the number of traps, since all the references between objects in the page will be converted when the page is read in. This relies on some degree of locality of reference and can be more or less successful depending on the data and the way objects are put together into pages.
- Pages previously brought in are periodically scanned for persistent addresses of objects which have since been brought in. The frequency of scanning has to be chosen with some care so that the time spent scanning is not greater than the saving in handling the faults.
- When a persistent store fault is taken other addresses "nearby" can be updated. Which addresses to look at depends very much on the structure of the code, but the savings can be considerable. For example the contents of all the registers can be updated, but the success of this will depend on the extent to which values are cached in registers.

8 Writing Back

At some point the user will want to write the changes that have been made in local memory back to the file. This basically involves a reverse of the process used when the objects were read in. The addresses in objects to be written out are converted back to their original persistent form and the pages are written, as described above, to new locations on the disc.

It is only necessary to write back objects which have changed and these will be a small proportion of the total. It would be possible to find out precisely which pages had changed by comparing the new version with the page on disc, but it is sufficient to assume that any of the pages of mutable objects may have changed, since there are not many of these.

Each address in an object to be written back must be converted to a persistent address. The maps are set up to make conversion of addresses from persistent to local form easy, but conversion the other way has to be done by searching through the maps for a local address. This can be made reasonably efficiently by having a simple cache of recently found local addresses and their persistent equivalent. Each entry in the cache refers to a page rather than an individual object in it so quite a small cache can be used.

Mutable objects are used for variables and arrays which can be updated. If a variable is set to refer to an object read from the file we will be able to convert the address back to the persistent store. However it is equally possible to create a new object in local memory and assign the address of that to the variable. This is what happens when new declarations are made in Poly or ML. Before the variable can be written back there must be a persistent address for the new object, so it must be turned into a persistent object, along with all the objects it refers to. The local objects are copied into pages by a process very similar to a copying garbage-collection. The addresses of the pages are put into free entries in the secondary maps and new secondary maps are made if there no free entries. It is possible to distinguish mutable and immutable objects by a flag bit so the two kinds of objects can be put into different pages.

Once the copies of the pages in store have been converted to the persistent form they can be written to the disc. They are written to previously unused blocks on the disc and to do this there is a bit map with a bit corresponding to each block. At the end of the writing process a new bit map is written out with the new locations of the pages shown as allocated and their previous locations now free.

In PS-Algol the system may write objects to the disc before *commit* is called when the memory becomes too full. In many database applications many different objects are used, each for only a short time. In these applications the memory

would quickly fill up with objects which were no longer required so PS-Algol has to have a mechanism to cope with this. In the Poly and ML system however objects, particularly procedures, tend to be used repeatedly or not at all. So far it has not been found necessary to clear objects out of the memory.

9 Garbage Collection

In normal operation once an object has been written out to the database it remains on disc even if it becomes unreachable. There is, however, a garbage collector which can be run periodically on the database to recover the lost space.

The garbage collector can be run in one of two modes, either non-compacting in which case it merely makes a new bit map of free blocks, or compacting when blocks at the end of the file are copied to free space nearer the beginning. In both cases the basic principle is the same. The garbage collector starts from the root of the database and finds all the objects that are accessible from it. Any page containing an accessible object is retained and the rest are garbage. The bit map and the entries in the maps which show the position of pages on disc are modified so that the garbage blocks on disc and their persistent addresses can be re-used. Working on whole pages and not compacting objects within a page means that persistent addresses do not change but it could lead to fragmentation over long periods.

The garbage collection process is basically the same as a mark phase of an in-store garbage collector but with one crucial difference. An in-store garbage-collector can usually assume that the cost of reading a word in memory is independent of its location whereas when garbage-collecting a disc the cost depends on whether the word is in a block which is in store or not. A simple recursive marking phase might result in a lot of disc activity. A better scheme is one where as many objects as possible in a block are processed together. This will reduce the number of times a particular block is read in.

The normal recursive scheme follows the addresses in an object as soon as they encountered, unless they refer to objects which have already been dealt with. There is one bit, the *mark bit*, which indicates that the object has been processed. It is used both by the marking phase itself to prevent an object being scanned more than once, and also as the result of the marking phase to indicate that the object is not garbage and must be preserved. The mark bits may be held with the object or in a separate bit map.

The scheme used in the disc block garbage-collector is a variation of this but uses a second bit map, the *pending bit*, to indicate that an address has been found but not followed. This bit is set when the recursive algorithm would recursively

process the object referred to. Using two bit maps allows the garbage-collector to work iteratively rather than recursively. Initially the pending bits corresponding to the root procedure of the database are set. The garbage-collector then reads a block which contains at least one object referred to by a pending bit, and ideally having several pending objects. Each of the pending objects in that block are now marked and removed from the pending map. All the addresses in these objects are added to the pending map, unless they refer to objects that have already been marked. Some of these addresses may be in the current block, especially if there is some locality of reference, and they can be processed now. When there are no pending objects left in that block another block is read in. This repeats until the set of objects to be scanned is empty. The blocks which were read in this process must be kept, the rest are garbage.

10 Multiple Users

The system described so far is for single users who have all their data in one database. Single user systems have the big advantage that all accesses are sequential and there is no need for any transaction mechanism. However the inability to share objects means that not only is a lot of disc space wasted with multiple copies of the same objects, but a user who produces something which is of use to others cannot pass it on. A mechanism for sharing data between users has been developed to reduce this problem.

10.1 Multiple Databases

One of the principles behind the design of the persistent storage system for Poly and ML was that it should be transparent to the users. What this implies for a shared data system is that the user should not be aware that some object is shared between several users, and that he should be able to use it as though he had exclusive use of it. In practice this is impossible to achieve without explicit control of concurrency so a compromise has to be made. In this system that is done by preventing shared variables from being updated, while allowing shared objects to be read.

The Poly system uses a scheme where each user has his own database which he can read or write. A user starts the session by executing the root procedure of his own database, the *root database* for that session. If a persistent address is used which refers to an object in another database it will be automatically opened, but only for reading. Calling the commit function causes variables read from the root database to be written back but other variables are not.

New databases are created by a special function which "spawns off" a new database with a new *root database*. Typically the root procedure will contain references to objects in the parent, and the parent will contain references to objects in the new database. In order to be able to follow these references it must be possible to find one database from the other, so each database contains a list of file-names of the databases to which it refers.

10.2 Implementation

It is implemented by using part of a persistent address as a *file number*. The address now consists of file number, page number within that file and an offset in the page. The file number is an index into a table of file names which can be found from the master map. Translating a persistent address may now involve opening a new file if the address refers to an object in a file which has so far not been used. It is necessary to modify the persistent addresses in objects read from other files because the file-number fields will index into the file name table of that file and will have to be changed into an index into the file name table of the current file.

10.3 Alternative Methods

This approach to sharing data has a number of problems. Perhaps the most serious is that the persistent store is no longer transparent since variables are only written back if they were read from the root database. The user can change variables read from other databases but those changes will not persist.

Another problem is the need for locking. One user may be using a database as their root database and periodically writing to it during a session, while another may be reading from it. The reader will have copies of values which have been read from the database in his own memory space and these may well be out of date. More seriously he will have copies of maps which give the disc addresses of pages on the disc. If the database has been updated several times it is quite possible that these maps will be out of date and the page at a particular disc address may be completely different to the one that was expected.

The only complete solution to implement a full transaction mechanism where objects are read in, modified and written out as a database transaction. As an object, or at least one which could be updated, is read from the database it would be locked so that it could not be read or changed by anyone else. The operation on it, either reading or writing, would be done, and the object would then be written back immediately and the lock released. This would be expensive since mutable objects cannot be held in store and every operation on them involves reading and writing to the database. An alternative would be to introduce the

idea of a transaction into Poly and ML so that locking and unlocking would be done explicitly. This might reduce the cost since locking would be done at a higher level, however the user must now become aware of the transaction system. On the whole it was felt that this would be too complicated and the solution adopted was to lock the database as a whole.

11 A Persistent Environment

The persistent Poly system has as its root a procedure which is called when the system starts. The root procedure calls through a procedure variable so that new procedures can be installed. This allows the user to decide, for example, whether to enter Poly or ML at the start of the session.

When using either Poly or ML the root procedure enters the read-eval-print loop for the language. This reads input from the terminal and sends it to the compiler. The compilers are pure functions operating on input and output streams and taking an *environment* as a parameter to maintain the state. In Poly the environment is a pair of procedures, one of which takes a string and returns a value, the other enters a string/value pair into the table. For ML the environment is rather more complicated because the name spaces for values, types, exceptions and infix status are distinct. Environments may be implemented in any way but a hash-table is convenient. The environment given to the compiler by the root procedure is part of that procedure's closure. During a session declaration of objects (procedure, values or types) made by the user are added to this table. When the session ends and the data are written back to the database the modified environment is written back as well. Hence the next session is run using the new environment and all the declarations made during the previous session are available.

An environment which maps names onto objects is very similar to a directory in a filing system so the environment system can be used as a form of typed filing system. The objects themselves are basically pairs of a value and information about its type. There is no reason why there should not be many environments available, some contained in others. This would correspond to a filing system which allows arbitrary directory structures to be built up. There is no requirement that the system of environments be a simple tree structure, an environment could contain a reference to itself or an environment pointing to it. Since the user can write his own environment in any way he likes he can incorporate any access controls he feels desirable. For example the environment could be written so that a password must be given before a function will return the environment.

12 Conclusions

Adding a persistent store system to Poly and ML was a fairly simple exercise, though work was needed to get it running efficiently. It would generally be the case that a persistent storage system as described could be added to any language in which all objects reside in the same memory space and are garbage-collected. It is also necessary for instructions to be restartable if the technique of using persistent addresses which cause memory traps is used.

The advantages of a persistent storage system over other methods of storing data are worth noting. Reading and writing the whole of a large core image is expensive and the overall size may be limited by the memory, real or virtual, of the machine. It is also impractical to explicitly read and write large numbers of objects.

Finally, the system has been in use for some time, and the ML implementation in particular is being used for developing large proof systems.

References

[Atk83] Atkinson M.P. et al. "An Approach to Persistent Programming". Computer J., Vol 26 No 4 1983.

[Atk81] Atkinson M.P., Chisholm K.J. and Cockshott W.P. "PS-Algol: An Algol with a Persistent Heap." Technical Report CSR-94-81, Computer Science Dept., University of Edinburgh.

[Bob67] Bobrow and Murphy. "Structure of a LISP System Using 2-Level Storage" Comm. ACM 10.3 March 1967.

[Cha78] Challis M.P. "Data Consistency and Integrity in a Multi-User Environment" In Databases : improving usability and responsiveness. Academic Press. 1978.

[Fal67] Falkoff A.D. and Iverson K.E. "The APL/360 Terminal System" Proc. ACM Symposium on Interactive Systems for Experimental Applied Maths.

[Ing78] Ingalls D.H.H. "The Smalltalk-76 Programming System – Design and Implementation" Proc. 5th ACM Symposium on Principles of Programming Languages. 1978.

[Mat85] Matthews D.C.J. "Poly Manual" SIGPLAN Notices. Vol.20 No.9 Sept. 1985.

[Mil84 Milner R. "A Proposal for Standard ML" in "Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming", Austin, Texas 1984.

Progress with Persistence in Poly and Poly/ML.

David C.J. Matthews

18 August 1987

Abstract

The paper describes the progress made on using persistence in the Poly programming language and some of the issues which still have to be resolved.

For some time Poly has been implemented on a persistent storage system which has allowed objects to be loaded transparently into store and modified objects to be written out again. This has proved to be a convenient way of allowing large systems to be developed without the overheads of loading the whole of a core image into store. It was however necessary for each user to have their own copy of the whole system since there was no way of sharing data between users.

A new version of the system has been developed in which databases can be shared between users. Each user has their own "home" database in which they keep their private data. They can also make use of data in shared databases which can only be read from. The system is arranged as a tree structure with a root database containing objects which are likely to be generally useful, and below that specialised databases, with the leaves of the tree being the users' databases. This structure allows databases to be opened automatically as persistent addresses are used so maintaining the transparency of the system to the users. The present scheme allows shared databases only to be opened for reading so that objects read from them are not written back. To this extent the behaviour is not transparent.

The paper also describes work on persistence in relation to processes and windows.

1 Background

There has long been a division in programming languages between those which are used in essentially a "batch" mode, where a source file is compiled into an object file, and the interactive languages, where the source text is compiled as it is typed in. The differences are in a sense more to do with the implementations than with the languages themselves and even in the interactive case there is usually a command to compile source text from a file. On the whole though languages tend to lend themselves better to one implementation or the other.

Apart from the way in which programs are constructed there is another difference between interactive and non-interactive languages. Non-interactive languages are compiled into separate programs and any communication between them must be through the operating system. This usually means through a text file or something similar. In contrast in an interactive session functions, or the equivalent, are applied to generalised data and generate data. The data may often be structured, such as lists or trees. Within a session this is easy to arrange, but we would often like to be able to suspend a session and come back later. This means that structured data have to be converted into a file that can be held in the filing system. The non-interactive system, which works on files anyway, does not have the same problem.

The conventional solution is simply to write out the contents of the memory at the end of the session and then read it back in at the start of the next[5]. This "workspace" or "core-image" idea allows a user to develop a system over a number of sessions. It has the advantage that the structuring and naming used is appropriate for the language rather than being imposed by the operating system. The disadvantage is that the facilities provided by the operating system for sharing between users is not available. The workspace has to be treated as a single entity and it is difficult to isolate a useful function from the workspace and make it available to other users, though[5] describes how a variable or function could be copied into the active workspace from another. In addition, since the whole workspace has to be read in to store there is the problem that it may grow too big to fit into the available memory. Solutions to this problem[3][7] have tended to require changes to the operating system rather than a system which would run on top of an existing system without modification.

2 Persistence

The idea of using persistent storage as a convenient way of implementing databases was proposed and implemented by M.P. Atkinson[1]. He noted that programs operating on databases often have two representations of data, one used internally

by the program while it is running, and a different representation for the same data when held in the database or filing system. The program was forced to spend a considerable time translating between the two representations, because the types provided by the database for persistent data were different from those provided by the programming language for transient data. His solution was to suggest that persistence of data should be a property which could be possessed by a value independently of any other property. In particular whether an object could persist was not associated with its type. The programmer could then choose a representation for the data purely on the basis of the algorithm and not in order to satisfy the requirements of a database system.

The language PS-Algol[2] was designed to test these ideas. A PS-Algol program can open and operate on data in a database, modify it and add new data to it. There are functions to open and close the database and to *commit*, or write back changes. These same operations would be present in any database system. The difference with PS-Algol is that there are no other operations to read or write data. Instead objects are read in as they are required. The programmer can use any data structures appropriate to the task and the storage system will ensure that objects are brought into store. In this way persistent data are treated in exactly the same way as transient data and can, for example, be combined together in a single structure. Changes are made by the normal assignment operation but are only recorded in the database when the explicit *commit* is called. The action of *commit* is to preserve in the database every object which could be found by following pointers from a number of distinguished roots. Since these roots are the only way in which a program can get access to the data this rule ensures that all the useful data are preserved.

A persistent store can be thought of as a cross between a virtual memory system and a database. Objects in a database can refer to other objects and any object can only be reached by following these references from a few well-defined roots. Transfers to and from the database are usually made by calls to special procedures. In a virtual memory system pages are transfered to and from backing store without any explicit requests from the user. They are regarded simply as unstructured store which, if it can be retained in the filing system at all, must be retained as a whole. A persistent store combines these ideas by having automatic transfers but retaining only reachable objects in the database.

Because objects are read from the database transparently the system behaves very much like one where the whole workspace is read in and written out at the end. Opening a database is similar to reading in the workspace and *commit* to writing it out again. The difference is that the cost of reading a persistent database is dependent on the amount of data used and not the overall size of the database.

3 Poly and Standard ML

Poly[8] and Standard ML[9] are general purpose programming language supporting polymorphic operations. They are both statically type-checked and statically scoped and treat closures as first-class objects. Their type systems are different but the underlying abstract machines are sufficiently similar for a common implementation to be used.

They are used interactively and in the Poly/ML system the two languages together comprise a single system. The original implementation of Poly, and other implementations of ML, have used the workspace idea to preserve data from one session to the next. When the size of the workspace became large an alternative was needed and a persistence storage system, based on that for PS-Algol, was designed.

The initial design of the persistent store for Poly and ML was similar to the PS-Algol work. However as the design progressed it became clear that there were properties of Poly and ML and the way they are used, that would affect the design.

It was regarded as important that the system should be transparent to the user as far as possible. The user should not have to think in terms of a database but in terms of the programming language and his own data structures. The persistent store in PS-Algol went most of the way towards this by making transfers of data from the database into store transparent. As far as the user of Poly or ML is concerned the system is very much like using a core image which gets read in at the start of the session and can be written out when required. The main difference is that the initial prompt appears almost immediately but there is a delay when the first command is executed as the compiler is brought into store. At any time the user can call the *commit* function which will write changes back to the disc. Changes are also normally written back at the end of the session.

Both languages are strongly type-checked and make clear distinctions between values which can be updated, *variables* or *references*, and those which cannot. These features are exploited in two ways. Since the type system will prevent addresses being used except as references to objects the system can operate on the addresses without the user being aware. This would not be possible in a language which allowed the user to extract the address of a word inside an object, for example. Distinguishing updatable, or *mutable,* objects makes it is possible to mark them when they are created so that the system can keep them separate from the non-updatable, or *immutable,* objects. In practice the vast majority of objects are immutable so allowing some optimisations which would not otherwise be possible.

The Poly and ML compilers are written in Poly and are part of the system. All the compiled code they generate and all the other data structures are also within

the system. Unlike in PS-Algol, where programs from outside can operate on the data and so there have to be ways into the data from outside, in Poly or ML the only operation needed is to start running the *read-eval-print* loop of either Poly or ML. The root of the database is therefore just a procedure which is called from the system when the session starts.

4 Multiple Users

The system described so far is for single users who have all their data in one database. Single user systems have the big advantage that all accesses are sequential and there is no need for any transaction mechanism. However the inability to share objects means that not only is a lot of disc space wasted with multiple copies of the same objects, but a user who produces something which is of use to others cannot pass it on. A mechanism for sharing data between users has been developed to reduce this problem.

4.1 Multiple Databases

One of the principles behind the design of the persistent storage system for Poly and ML was that it should be transparent to the users. What this implies for a shared data system is that the user should not be aware that some object is shared between several users, and that he should be able to use it as though he had exclusive use of it. In practice this is impossible to achieve without explicit control of concurrency so a compromise has to be made. In this system that is done by preventing shared variables from being updated, while allowing shared objects to be read.

The Poly system uses a scheme where each user has his own database which he can read or write. A user starts the session by executing the root procedure of his own database, the *root database* for that session. If a persistent address is used which refers to an object in another database it will be automatically opened, but only for reading. Calling the commit function causes variables read from the root database to be written back but other variables are not.

New databases are created by a special function which "spawns off" a new database with a new *root database*. Typically the root procedure will contain references to objects in the parent, and the parent will contain references to objects in the new database. In order to be able to follow these references it must be possible to find one database from the other, so each database contains a list of file-names of the databases to which it refers.

4.2 An example

This scheme has been used in a practical application for a theorem proving system, Isabelle[11], written in ML by L. Paulson. The basic Isabelle system is independent of any particular logic and is parameterised by a logic in order to be able to prove theorems using it. There are at present two logics for Isabelle, one using Martin-Löf type theory and the other using a set-based logic.

The system has been set up using the database heirarchy with a system database at the root. This contains the Poly and ML compilers and also standard types such as *int* and *list*. Below that is a database with the unparamaterised Isabelle system. Below the Isabelle system are two databases, one containing the type logic and the other the set logic. Finally there are the users' databases below these.

Each user still has their own database, but these are now relatively small since they contain only their own data. They can make use of declarations in the parents as though they were in their own database.

Since the parent databases are read-only they can be used in a distributed system. They are held on a shared virtual disc which can be mounted by users on different machines on a network who have their own databases on local discs.

4.3 Alternative Methods

This approach to sharing data is particularly suitable for large data structures which are read-only. It has a number of problems when writing to shared data. Perhaps the most serious is that the persistent store is no longer transparent since variables are only written back if they were read from the root database. The user can change variables read from other databases but those changes will not persist.

Another problem is the need for locking. One user may be using a database as their root database and periodically writing to it during a session, while another may be reading from it. The reader will have copies of values which have been read from the database in his own memory space and these may well be out of date. More seriously he will have copies of maps which give the disc addresses of pages on the disc. If the database has been updated several times it is quite possible that these maps will be out of date and the page at a particular disc address may be completely different to the one that was expected.

The only complete solution to implement a full transaction mechanism where objects are read in, modified and written out as a database transaction. As an object, or at least one which could be updated, is read from the database it would be locked so that it could not be read or changed by anyone else. The operation on it, either reading or writing, would be done, and the object would then be

written back immediately and the lock released. This would be expensive since mutable objects cannot be held in store and every operation on them involves reading and writing to the database. An alternative would be to introduce the idea of a transaction into Poly and ML so that locking and unlocking would be done explicitly. This might reduce the cost since locking would be done at a higher level, however the user must now become aware of the transaction system. On the whole it was felt that this would be too complicated and the solution adopted was to lock the database as a whole.

5 Processes and Windows

As part of the basis of a programming environment a window package for Poly and ML has been developed. It provides the normal drawing operations for lines and text and for displaying bitmaps. It also allows for input from the keyboard or a mouse.

Handling activity on windows is most conveniently done by having a separate process manage each one. A process package was implemented for Poly and ML using ideas from CCS[10] and PFL[6]. This provides functions for forking processes and sending and receiving data on channels.

As with other objects in the Poly system processes are persistent, requiring the process state to be retained in the database. Most of the state is held in the process's stack and this can be written out to the database to save the process. A single stack segment is used for each process since it simplifies function calls but the segment is arranged so a small segment can be used initially and the segment will grow as required.

Persistent processes make possible a change in the way the database is written. Rather than having as the root of the persistent store a procedure which is called at the start of each session the *commit* operation now saves the list of active processes. Anything reachable from an active process is retained in the database. When the database is opened these processes are reactivated. The state is then the same as it was when *commit* was called. Normally there will be a process executing the read-eval-print loop for either Poly or ML.

Saving the list of active processes also allows the contents of windows to be saved, or more accurately, for them to be redrawn when the database is reopened in the same way that they appeared when it was committed. A window on the screen may need to be redrawn at any time because part of it has become exposed. When this happens the window package provides an event on the event stream of the window so that the program can redraw it. Because this can happen at any time there must be a process prepared to do the redrawing. The process can be saved

and then reactivated in a new session. If it is now told that the window needs redrawing it will draw the contents exactly as they were in the last session. This is an active way of retaining data. Rather than save an image of the screen the process for drawing the data is retained.

References

[1] Atkinson M.P. et al. "An Approach to Persistent Programming". Computer J., Vol 26 No 4 1983.

[2] Atkinson M.P., Chisholm K.J. and Cockshott W.P. "PS-Algol: An Algol with a Persistent Heap." Technical Report CSR-94-81, Computer Science Dept., University of Edinburgh.

[3] Bobrow and Murphy. "Structure of a LISP System Using 2-Level Storage" Comm. ACM 10.3 March 1967.

[4] Challis M.P. "Data Consistency and Integrity in a Multi-User Environment" In Databases : improving usability and responsiveness. Academic Press. 1978.

[5] Falkoff A.D. and Iverson K.E. "The APL/360 Terminal System" Proc. ACM Symposium on Interactive Systems for Experimental Applied Maths.

[6] S. Holmström. *PFL: A Functional Language for Parallel Programming*, Proceedings of the Declarative Programming Workshop, University College London, 1983.

[7] Ingalls D.H.H. "The Smalltalk-76 Programming System – Design and Implementation" Proc. 5th ACM Symposium on Principles of Programming Languages. 1978.

[8] Matthews D.C.J. "Poly Manual" SIGPLAN Notices. Vol.20 No.9 Sept. 1985.

[9] Milner R. "A Proposal for Standard ML" in "Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming", Austin, Texas 1984.

[10] Milner R., *A Calculus for Communicating Systems*, Lecture Notes in Computer Science, no 81, Springer Verlag, 1981.

[11] Paulson L.C., *Natural Deduction as Higher-Order Resolution* J. Logic Programming 1986:3 237-258.

An Implementation of Standard ML in Poly

David C.J. Matthews

1 Introduction

This document describes an implementation of Standard ML [1] written in Poly [2]. This is a complete implementation of the language including the modules system.

The original aim of the implementation was largely as an experiment in using Poly for implementing reasonably sized programs, and only secondarily to produce a working ML system. In fact, though, the Poly/ML compiler has been very successful and is in use for a number of serious projects.

This description is mostly aimed at those interested in the details of the implementation or planning to implement their own compiler for ML or a similar language. It includes only the parts of the system specific to ML. The low-level system is described in a separate report.

2 Overview

The compiler can be divided fairly neatly into two main parts, the front-end which translates the ML source into a machine- and language-independent code-tree, and the back-end which optimises this code and translates it into machine-code. The back-end is shared between the Poly and ML compilers.

The ML front-end takes three passes to get to the intermediate code. The first pass parses the source text and generates the parse tree. It deals with infix precedences and type variables, but does no other identifier matching, in particular it does not distinguish between variables and constructors. The second pass matches up identifiers to their declarations and does the type checking. The third pass generates the intermediate code. It uses type information provided by the previous pass to resolve overloading and to do the completeness checking for patterns.

The back-end consists of an optimiser which takes the code tree from the front end and generates an optimised copy for the code-generator. It recognises tail-recursion and procedures for which a full closure need not be made. The other optimisations, such as inline code expansion and handling type-vectors, are intended for code from the Poly compiler. The machine-code generator compiles the code into vectors in store. Currently there are code-generators for the Vax, for the Sun2 and Sun3 (68010 and 68020) and a version which generates an interpretable code. Other code-generators may be written in the future. This part of the system is described in a separate document.

The compiler is divided into several modules which implement different data structures and operations on them. When the compiler modules are composed the result is a function with a Poly type equivalent to:

```
(unit->string) * (string->unit) * environ -> unit -> unit
```

that is a function of three arguments which returns a function. The first argument returns the next character from the input stream and the second is used to write out error messages. `environ` is a tuple of functions (in Poly a *type*) which enter and look up the different classes of values (types, values, infix status, exceptions, structures and signatures). It is used both to look up names in the outer scope during compilation and to enter the names when the code is executed. The result of compilation is a function which, when called, executes the code and "side-effects" the environment. In normal use this function is executed immediately the code is compiled.

The compiler compiles only one top-level phrase and returns a function which will execute it. If it finds any errors in the phrase, either in the syntax or perhaps a type mismatch, it raises an exception.

The basic compiler module is not used directly on files or from the terminal. Instead a "shell" procedure is wrapped round it which handles input from and output to the terminal. A separate procedure is used for compiling files. If there is an exception either from compilation errors or through the compiled code raising an exception the procedure for compiling files exits immediately and does not try to compile any more top-level phrases, whereas the interactive procedure prints out the name of the exception and then continues.

Other top-level procedures are possible, for example the *make* function uses a special environment to compile a program consisting of several modules by only compiling the modules that have changed since the last compilation. It works by having a special look-up function which checks to see whether a module is up to date and recursively calls the compiler.

3 The Parser

The parser reads ML source text and generates the parse tree. It is structured as a lexical analyser module *lexan* and parser modules *parse_dec*, the main expression and declaration parser and *parse_type* for types.

3.1 The Lexical Analyser

The lexical analyser module is fairly conventional. The input stream to the compiler is used by the lexical analyser to return the next lexical symbol. The main point to note is that names such as `A.b.+` which are structure selections, are returned as single entities and are separated out later on. Reserved words are looked

up at this point. Symbols such as `=`, `*` and `->` which can be treated as identifiers in some circumstances, are passed back as keywords.

The lexical analyser also contains the error-message routine. Error messages are produced using an Oppen-style [4] pretty printer. This is particularly useful because the parse-tree can be unparsed to give some context information. Generating an error message also sets a flag to abort the compilation at the end of the current pass.

3.2 Parsing Expressions and Declarations

The parser is a fairly conventional recursive-descent parser. This was chosen in preference to using a parser-generator such as YACC partly because of the problems of handling declarations of operator precedence but also because a parser-generator for Poly was not available.

The parser converts infix declarations into their equivalent prefix form and to do this it has access to the environment of infix precedences. Operator precedences in ML are under the control of the user and can be changed by local declarations. For example

```
let infix 8 + in 1+2*3 end
```

returns the value 9 because the `+` has been given a higher precedence than `*` for that expression. The parser must have access to global precedence declarations and know about the scope of declarations. This is quite easy with a recursive-descent parser since the parsing structure is represented in the recursion so all that is required is to pass the current scope table as an extra parameter to the parsing procedures.

Parsing an expression involving infixes is done by a recursive method. A simple way of doing this would be to have a procedure which recurses once for each precedence level and handles all the operators at that level. This is very inefficient since all expressions will require at least 10 procedure calls in order to reach the atomic expressions such as identifiers and constants. A much more efficient method is to recurse only when a more binding operator follows a less binding one. For example, the expression `a*b+c=d` can be parsed without recursing because each operator is followed by one with a lower precedence. The reverse-Polish representation is simply `a b * c + d =`. A slightly more complex expression `a*b+c=d+e` requires one recursion to handle the right-hand operand of the equality but no more. Right-associative operators can be handled by this scheme simply by recursing if the operator is right-associative and followed by an operator of the same precedence[1].

Since the parser has to handle infix declarations in order to parse an expression it is sensible if it is also responsible for passing top-level infix declarations back to the global environment. It is possible for top-level declarations involving infixes to

[1]This is slightly more difficult than it might be because in ML two different operators of the same precedence always associate to the right, even if they are both declared as right-associative.

fail if several declarations are combined in a "local-declaration" so top-level infix declarations are first put into a temporary environment until the compilation and execution are complete when they can be put into the real global environment.

The parser also matches up explicit type variables since these have fairly simple scope rules, but it does not attempt to match up other identifiers with their declarations because of mutually recursive declarations.

The classes of identifiers accepted depends on the context. In particular = can be used in an expression but cannot be declared. The reserved type constructors * and -> can be used as ordinary identifiers.

The expression and declaration parser calls procedures in the "parsetree" module to make the parse tree. The parse tree is returned as the result.

Parsing a pattern is complicated by the syntax of a layered pattern.

```
pat ::=
apat
pat : ty
<<op>>var<<:ty>> as pat
```

This is ambiguous because a variable can also appear in an atomic pattern (apat). The solution to this is to change the syntax to

```
pat ::=
apat
pat : ty
pat as pat
```

and parse it that way. The check that the first pattern is just a variable is then done on the parse tree.

Constructors and variables are not distinguished in the parser and this makes parsing clausal functions, in particular, rather more difficult. The declarations

```
infix %;
fun (a :: b) % c = ...
fun (a%b) c d = ...
```

both declare % as a function. To find the name being declared it is necessary to unravel the infix application. The simple solution is to parse each clause as though it were a single pattern and the syntax were

```
fun pat = exp — ... — pat = exp
```

Since the parser makes no distinction between constructors and variables this actually parses the same syntax as the full definition. The business of extracting the name of the function is then left to the second pass.

3.3 Parsing Types

Type expressions are parsed a separate module since the syntax is completely separate from that of expressions and declarations. The main concern here is that type constructor names must not include =, * or ->. The result of parsing a type expression is a type tree generated by the *typetree* module.

4 The Parse Tree

The parse tree, or more accurately the abstract syntax tree, is at the heart of the compilation. The parsing routines call construction functions to make the tree and the result of parsing a top-level declaration is this tree.

4.1 Internal Structure

Entries in the parse-tree can be divided into classes: types, declarations, patterns and expressions, for example. The syntax of ML allows them to be combined in limited ways, a declaration appears in an expression only as part of a **let** declaration, so it would be possible to represent the parse-tree of a declaration with a different parse-tree type than an expression. In practice it is convenient to distinguish only between the parse-tree for types, *typetree* the parse-tree for structures and signatures *structs* and that for everything else, *parsetree*. Splitting this way can be made without circular dependencies since a type cannot contain an expression or a structure and an expression cannot contain a structure. All the type-checking code forms part of *typetree* while most of the intermediate code generating procedures are in *parsetree*.

4.2 Structure

The tree is close enough to the original source text that it can be unparsed into a recognisable form to generate error messages. Each parse-tree type is a union of records with a separate record type for each abstract syntax form.

5 The Second Pass

Each of the parse-tree modules contains procedures used in the second pass. The second pass is responsible for matching identifiers to declarations and doing the type checking. The core of the second pass is a procedure in *parsetree* which recurses down the tree and returns its type. The environment is updated by declarations so a side-effect of the second pass is the declarations. Identifiers are represented as variables which are initialised to an "undefined" value when they are created by the parser and then assigned a pointer to the actual declared variable.

5.1 Unification

Type-checking in ML is done by unifying type expressions. A type is represented as an expression involving terms which are either constants, such as `int`, operators such as `->` or `list` and type-variables such as α, written in ML as `'a`. To match two types the type expressions have to be combined so that a common type can be made from them. As a side-effect some type-variables may acquire values. For example, the *append* function has a type `'a list * 'a list -> 'a list`. This means that it can be applied to two lists and produces a list but that the elements of the list are the same in each case. Suppose this is to be applied to a the value `(nil, [1])` , a pair of values with the first being the nil value and the second a list containing a single integer. This pair has type `'b list * int list` where `'b` is a different type variable to `'a`. When *append* is applied to the pair the type expressions `'a list * 'a list` and `'b list * int list` are unified. The operators `*` and `list` must match exactly but the type variables will match anything. The first occurrence of `'a` matches `'b` so `'a` and `'b` become the same type variable. The second occurrence of `'a` matches `int` so this combined type variable now takes on the value `int`. This is a constant which will only match other occurrences of `int`. The result of this is that `'a`, `'b` have both acquired the type `int` so the result of the application of *append* has type `int list`.

When types are represented as trees with the operators as nodes and the constants as leaves the type-variables can be represented using updatable references (variables). Unification of two expressions is then simply a matter of checking that constants and operators are the same in both expressions and where type-variables occur in one expression assigning the corresponding tree in the other expression into it. When two type-variables occur at the same point in both expressions either type-variable may be assigned to the other.

Unfortunately there are a number of complications which have to be dealt with before this scheme is satisfactory. Any number of type-variables can be unified together to make a single type-variable. In the example above `'a` and `'b` were unified and we have said that this results in one being assigned to the other. Since there is no difference between them suppose this resulted in `'a` pointing to `'b`. A little while later `'a` is unified with `int` which would result in `'a` now pointing to `int`. However `'b` must also point to `int` so rather than overwrite the pointer in `'a` we chain down the type-variables and point only the last one at `int`. This is shown in figure 5.1. The result of this is that the occurrence of a type-variable in a tree does not immediately mean that it will match anything. It is only after searching to the end of the chain of assignments and finding that the final entry is unset that we can conclude that it really is a type-variable. If it assigned to a constant or an expression then that becomes the value to be unified. As an optimisation all the type-variables in a chain can be pointed directly at the final entry to save following the chain repeatedly.

The next complication to be overcome is what is known as the "occurs" check. Suppose we are type-checking the following expression.

Initially.
'a ⟶ <unset>
'b ⟶ <unset>
After first unification.
'a ⟶ 'b ⟶ <unset>
After second unification.
'a ⟶ 'b ⟶ int

Table 1: Unification of 'a, 'b and int

```
fn x => x ::  x
```

This is a lamba-expression with bound variable x and whose body is the list constructed from using x as both the head and the tail. This is of course a type-error because x would have to have both a list type, for the tail, and the type of the element of that list, for the head. Detecting the error is quite difficult. Bound variables are given a type which is a new type variable, call it 'x. If the rules described above are followed then this would result in 'x pointing at 'x list, in other words a circular structure. To avoid this we must check when a type variable is assigned to an expression that the expression does not contain the type variable anywhere.

A further complication has to do with what are called *generic* and *non-generic* types. Lambda-bound variables, such as x in the previous example, have the same type everywhere they occur, that is 'x was used for the type of both occurrences of x. On the other hand a function such as *append* can have a different type at different occurrences. It may be applied to lists of integers in one place and lists of strings in another.

This is done by copying the type of the function each time it is used and making new type-variables for each of the old ones, being careful to share the type-variables correctly. For example each instance of the type of *append* must look like 'a list * 'a list -> 'a list and not 'a list * 'b list -> 'c list which would result if a new type-variable were made every time a type-variable was encountered. The *copy_type* procedure which does the copying keeps a list of each type-variable in the original structure and the corresponding new type-variable and only makes a new one if it finds one which is not in the list.

A lambda-bound variable has the same type everywhere but a function such as *append* can have different types in different places. How do we know whether to copy a type expression or not? A simple mechanism would be for type-variables belonging to lambda-bound variables to be marked as "not-copyable" so that *copy_type* would put them in the result rather than copying them. Otherwise they would behave like normal type-variables. Whenever an identifier is found its type is submitted to *copy_type* to get its effective type and in this way lambda-bound variables would have the same type at each occurrence, but generic variables would have their type variables copied. When the function has been compiled the

type of the result would have to be processed to convert all the "not-copyable" type-variables into "copyable" ones so that function is generic.

This works only provided lambda-bound variables are not unified with other type-variables. Unification of a "not-copyable" type-variable with a "copyable" one can be arranged so that the copyable one points to the non-copyable one, the result being a non-copyable type-variable, but suppose we have to unify two non-copyable type-variables. For example

```
fn a => fn x => a
fn a => let val f = fn x => a in f end
```

These two lambda-expressions are both the same, they take a value and return a function which returns that value. The type should be `'a -> 'b -> 'a`. The only difference is that in the second case the inner lambda-expression is bound to an identifier before being returned. Suppose the type-variables corresponding to `a` and `x` are `'a` and `'x` then the type of `fn x => a` would be `'x -> 'a`. However what is the type of `f`? Our simple-minded scheme would say that at the end of compiling `fn x => a` the type-variables are made copyable so that `f` would have a type `'p -> 'q` where `'p` and `'q` are new copyable type-variables. This is wrong because while `'x` should be made copyable `'a` which is still in the scope of the outer lambda, should not. The correct type for `f` is `'p -> 'a`. To get this correct type-variables are given a level number and the procedure to make the type-variable copyable only makes them copyable if they are above a certain level. When type-variables with different levels are unified the higher number is set to point to the lower. In this way at the end of compiling `fn x => a` the type will be `'x -> 'a` and `'x` with level 2, say, will be made copyable, but `'a` with level 1 will not. It will only become copyable at the end of the outer lambda-expression when level 1 type-variables are made copyable.

5.2 Constructors

Type constructors are processed in the second pass and value constructors and equality functions are made for datatype declarations. Datatypes with only one constructor do not require a union to be made so the value constructor is a null operation. If there is more than one constructor then a tagged union is made. Tagging can be avoided if the values in the union are disjoint. Luca Cardelli [3] describes a scheme where values which are addresses can be distinguished from those which are not, allowing an efficient implementation of datatypes like lists and trees. At the moment the only optimisation is that datatypes consisting only of constants, essentially an enumerated type, are treated specially. The built-in constructor "list" is a special case.

6 The Third Pass

The third pass generates the intermediate code tree from the parse tree. It does some checking which cannot be done in the second pass. The result of the third pass is code which produces a vector containing the values and exceptions declared in the top-level declaration.

6.1 Completeness and Irredundancy Checking

Checking for the completeness and irredundancy in a set of patterns is done during the third pass largely because knowing that a match is complete allows somewhat better code to be generated. The method used is due to Alan Mycroft and was used in the Edinburgh ML compiler.

The algorithm can be thought of as a process of covering an area, the *floor* with *tiles* of different shapes and sizes. Each pattern in a match corresponds to a different tile and together the tiles must cover the floor if the match is to be exhaustive. The irredundancy check is represented by saying that each new tile layed must cover some of the floor which was previously exposed.

The basic kinds of patterns to be handled are wild cards, constructors and pairs. Variables are treated as wild cards since they match the whole of a tile. Integer and string constants are a special kind of constructor which unlike other constructors cannot make a complete match by enumerating all the possible values. Tuples with more than two elements are represented by pairs of a value and a pair. Labelled records are treated like tuples.

The completeness checking procedure takes an area of floor and a pattern (a tile) and returns the area of floor left after the tile has been layed. Initially the floor is *all_there,* if the match is complete the result should be *all_gone.* At intermediate stages some of the floor area may have to be blown up so that part of it can be removed. Suppose we had the following set of patterns in a match.

```
a :: nil
nil
x :: y
```

Initially the floor is *all_there* which will be represented by "...". The first pattern to be matched is `op :: (a, nil)` in prefix notation, that is the constructor `::` applied to a pair. This does not match the whole of "..." so it must be expanded into the possible constructors. The match becomes

```
:: (...)
nil (...)
```

In other words the two possible constructors, each applied to *all_there.* For convenience all constructors are treated as having one argument, constants such as `nil` in patterns are treated as though they were `nil (_)`. We can now attempt

to remove the :: entry from the list by matching its argument. The argument in the pattern is a tuple so the argument in the floor must be similarly expanded.

:: (..., ...)
nil (...)

This applies only to the particular sub-tile being matched, so the argument to `nil` is not affected.

Pairs are matched using the identity

$$(x \times y) \setminus (a \times b) = (x \setminus a) \times y \cup x \times (y \setminus b).$$

The two special cases are detected. Where $x \setminus a$ (or $y \setminus b$) results in a complete match the $(x \setminus a, y)$ (or $(x, y \setminus b)$) term is removed. The other special case is when $x \setminus a = x$ ($y \setminus b = y$) when the result is the original tile (x, y). This can result in an explosion of areas of floor since two terms are produced for each pattern, and in particular the sub-term $(x \setminus a) \times (y \setminus b)$ is present in both the resulting terms, but in practice most patterns fit into one or other of the special cases so the tiles do not grow too complex. This identity is also quite easy to program since the $x \setminus a$ expressions are just recursive calls. An alternative, such as

$$(x \times y) \setminus (a \times b) = (x \setminus a) \times b \cup x \times (y \setminus b),$$

which would avoid the common sub-term, is actually more difficult to program. This is because it requires b, which represents a tile, to instead represent an area of floor.

In this example the two terms produced by matching `(a,nil)` are `(∅, ...)`, since the identifier `a` matches everything, and `(..., ::(...))` after the second `...` has been expanded into `nil(...)` and `::(...)`. The first of these reduces to ∅ and is removed, leaving the resulting tile as

:: (..., :: (...))
nil (...)

The next pattern, `nil`, removes the second entry, and the final pattern removes the first. The final result is an empty set and the match has been shown to be exhaustive.

6.2 Matches

Matches are code-generated by processing each pattern twice, first to generate a test to see if the value matches the pattern, and then to generate code to extract the values of variables. The reason for not doing these both together is that the code tree being generated can return only a single result. It would be possible to combine the passes and raise an exception if a match did not work, but on the whole this does not seem worthwhile. A better solution is to generate more efficient testing code by making a tree of patterns so that each constructor is tested only once and the leaves would then be the expressions corresponding to each pattern.

6.3 Exceptions

Exceptions in ML have a scope just like other values, but by their nature, they may be declared, be raised but then propagate out of the scope of their declaration. They cannot then be caught except by the wild-card handler. It is possible to construct pathological examples involving recursive declarations and functions declared in the scope of an exception which then pass out of its scope which essentially mean that an exception must be treated as a run-time value.

An exception declaration generates-code to create a unique value by creating a one-word value on the heap. A handler for an exception then has to do a simple pointer equality test to see if an exception matches. This works for ML but needs a slight modification so that it is compatible with code from the Poly compiler. Exceptions in Poly are represented by a string and do not have parameters, so raising an ML exception involves saving the exception identifier and the parameter away in global variables, raising a Poly exception with the exception name as a string, and then taking out the identifier and parameter when the exception is handled. The string is useful for identifying uncaught exceptions at the outer level.

6.4 Equality Testing

The = function, along with `print` and `makestring`, is special in that it is infinitely overloaded. Each new datatype that is declared can give it a new meaning. `print` and `makestring` are defined on all types, but the result of printing a function is just "fun". = on the other hand is not defined on functions or on type variables.

A simple way of handling equality would be to have a procedure which compared any two structures by using the contents of structures alone. This assumes that there is enough information in the values to distinguish addresses from data which must be there for the garbage-collector. It must also be able to distinguish references from other data because references are only regarded as equal if they are actually the same reference. This procedure could be used to compare values of any type without the type information.

It is however necessary to check at compile-time whether equality testing is allowed on values of the particular type. For example,

```
fun f x = (x = nil)
fun f p = (p = op +)
```

are both illegal, in this first case because `nil` has type `'a list` and so contains a free type variable, and the second because it involves comparison with a function. The first case would be legal if a type constraint, such as `(x: int list)` , had been used.

Equality testing is allowed for `X list` provided that it is allowed for `X`, because the definition of `list` as

```
datatype 'a list = nil | op :: of 'a * 'a list
```

contains no free type variables or functions. It would be possible to check this on every occurrence of `list` in a comparison, but it is more sensible to check this once when the datatype is declared. Comparing values of a type construction such as `int list` is allowed if it is allowed for each type constructor in it. Actually seeing if comparison is allowed for a type constructor is not trivial if there are mutually recursive type declarations where comparison may be allowed for one type only if it is allowed for another.

The result of this checking is a flag in each type constructor showing whether values of the type can be compared. It is not difficult to extend this slightly so that the flag is actually the code for a function which does the comparison with a special value which represents "comparison-not-allowed". For nullary type-constructors like `int` this is quite easy, but it is slightly more difficult for constructors like `int` which are polymorphic. How values of type `X list` are compared depends on the comparison function for `X`. The code actually generated for the type constructor `list` is roughly equivalent to

```
fun eq_list eq_X=
                fn (a::x, b::y) = eq_X(a,b) cand eq_list(x,y) |
                   (nil, nil)   = true |
                   _            = false
```

7 Modules

The definition of modules in ML is still under development so this can only be a description of the design as it is at the moment. The current design is based on the *structure*, a group of declarations packaged together; a *signature*, which is the "type" of a structure; and the *functor* which is a structure that can be parameterised by other structures. None of these can appear inside expressions or declarations so the module mechanism can be largely added on top of the core language system. The main exception is that names can be qualified by the name of the structure which contains the declaration and qualified names can appear anywhere. There is also an "open" declaration that can be used to make the names in a structure available without having to qualify each of them with the structure name. This can be used anywhere a declaration can appear.

8 Abstract Machine

The code generated by the third pass is a code tree which contains instructions and data but the flow of control is given by the tree itself. Using a tree makes global optimisation and inline code insertion somewhat easier than with segments of code, though the representation does take rather more store.

The code does not contain many of those instructions for manipulating data, such as adding together two integers, which are normally considered as part of

an instruction set. Instead these operations are treated as calls to functions in a standard vector. A simple code-generator would generate these as ordinary function calls but the code-generators for the Vax and 68020 both recognise special entries in the vector and generate the appropriate instruction in-line.

9 Machine Code Generation

The final phase of compilation is to generate the machine code. The only code-generator available at the moment is for the VAX but other code-generators may be written in the future. They are all likely to be fairly similar. On the whole the code-generator is similar to that for a language like Pascal, but there are a few differences. The main difference is that variables (references) are not held on the stack. This means that a value on the stack cannot change and so can be cached in a register without too much difficulty.

The VAX machine code is based on bytes and an instruction can start at any byte in the word and occupy almost any number of bytes depending on the arguments. The code is generated into a vector of bytes which can expand as necessary and is then copied into a code-vector of the correct size at the end of code-generating the function.

9.1 Linkage Conventions and Register Usage

The VAX provides instructions for procedure call and return which set up and remove stack frames and save and restore registers. Apart from the program counter and stack pointer registers which have special uses, two registers are designated as frame pointer and argument pointers and are used by the standard call and return instructions. The standard linkage conventions are fairly powerful but do not provide any support for a static link to get access to free variables. They are also fairly slow which is a serious problem for a functional language where function call must be fast. For these reasons a very light-weight linkage convention is used which uses the subroutine call and return instructions.

The conventions differ slightly according to whether the function is being called with a static link or as a full closure. In the case of static link call register 1 contains the static link and the subroutine jump is to the start of the code. For a closure call the register contains the address of the closure and an indirect jump is made using the address in the first word of the closure as the destination. The arguments to the function are passed on the stack so after the call the stack contains the return address, pushed by the subroutine call, and the arguments. No other values are explicitly put on the stack. Register 1, containing either the static link or the closure address, is marked on the pseudo-stack and will be pushed onto the real stack if the register is required, in particular for another function call, in same way as any other register. Small functions which do not do any function calls themselves, never need to push register 1 and any references to variables in the

closure can be done by indirecting through the register.

The code-generator keeps a note of the number of items on the stack and addresses all the values on it using the stack pointer register. This avoids the need for a frame pointer register but makes reading the assembly code listing somewhat difficult because the offsets of local values and arguments change according to the number of items currently on the stack.

The only use made of the frame pointer register in the code is in dealing with exceptions. In the standard VAX convention the word at the base of the frame is the address of a procedure for handling exceptions and it is initialised to zero by the standard procedure call instruction. The Poly/ML convention treats this word, the exception location, somewhat differently. When a handler is set up two words are put onto the stack, the old value of the exception location and the address of the new handler. The exception location is then set to the value of the stack pointer, in other words to the address of these two words. If an exception happens the stack pointer is reset to the value in the handler and these two words are popped from the stack, one back into the exception location to reinstate the previous handler, and the other into the program counter to execute the handler. If control reaches the end of the scope of an exception handler without the handler itself being executed the previous handler is reinstated and the handler address is removed from the stack.

References

[1] Robin Milner, *The Standard ML Core Language*, Edinburgh University.

[2] David C.J. Matthews, *The Poly Manual*, SIGPLAN Notices, September 1985.

[3] Luca Cardelli, *Compiling a Functional Language*, 1984 ACM Symposium on Lisp and Functional Programming.

[4] D. C. Oppen, *Prettyprinting*, ACM ToPLAS Vol. 2 No. 4 Oct 1980.

The Poly and ML System: Abstract Machine and Realisation

David C.J. Matthews

17 September 1987

1 Introduction

This document forms part of a set describing the Poly[3] and Standard ML[2] systems. It describes the abstract machine and its realisation on the Vax and the MC68020 and on a interpreted byte-code system.

2 Overview

The Poly code-generators generate segments of code from a tree representation. The tree is constructed by the Poly and ML front-ends but passes through machine- and language-independent global optimiser and pre-code passes before it gets to the code-generator. The code-generator for the interpreted system essentially flattens the tree into a byte-code so rather than describe the tree structure and its mapping into the various machine-codes, the byte-code system will be used as a model for the other two versions.

3 Basic Byte-Code System

The byte-code system is a simple stack machine in which each instruction is a single byte, possibly followed by a number of operand bytes. It has only a few primitive instructions, see Table 3, so there is plenty of scope for using the remainder of the 256 available opcodes for shorter versions of common cases. For example, in addition to having an instruction to load a full-length immediate constant it might be worth having an instruction to load a single byte constant to be used for

values less than 256. Going one step further it is possible to combine frequently used operand combinations with the opcode, so that, for example an instruction to load the constant value zero would be a single byte. These optimisations will be described in due course.

The machine has only four registers, the stack pointer, the program counter, the handler register and a temporary register. The stack pointer, *sp*, points to the top entry on the stack which is used for instruction operands, function parameters and results, return addresses and for stacking exceptions. The program counter, *pc*, points to the current instruction in the code. The handler register, *hr*, points into the stack at the current exception entry. The temporary register, *t*, is only explicitly used during exception handling.

A single stack is used on which procedure parameters, local values, link information and exceptions are held. A procedure call involves first pushing the arguments onto the stack and then executing a call instruction which pushes two words of link information. Local declarations inside the procedure also result in values being pushed onto the stack, as does setting up an exception handler. Intermediate results are treated as local declarations, indeed the distinction between local declarations and intermediate results is arbitrary since the higher-level parts of the compiler may remove local declarations made by the user. for example where the value is a constant, and introduce extra declarations as a result, say, of expanding inline functions. Values are popped from the stack when they are no longer needed. There is no frame pointer register so values on the stack have to be addressed relative to the stack pointer. All the objects on the stack have fixed size so the code-generator knows the number of words on the stack at any time and can work out the offset. Not having a frame pointer means that procedure call and return is faster since there is no need to save an extra register. In the machine-code versions it also saves instructions which is perhaps more important.

The most obvious point is that the machine does not have any instructions for operating on data, such as adding integers, or reading from a file. These are all done using the standard closure call mechanism, but with special closure addresses. In the byte-code system these are trapped by the interpreter but in the code-generated versions they are the addresses of code in the run-time system.

This has several advantages. First of all, in a functional language, the operation to add two integers is a true function and can be passed around or stored in variables in any way that satisfies the type system. With this mechanism, addition behaves like any other function until it is applied.

It is still possible for the code-generator to implement an addition instruction by recognising the case of a function application when the function is a constant whose value is the special address for addition. This, however, is an optimisation, and the choice of whether to make it or not depends on the expected frequency of

jump	Unconditional jump.
jump_false	Pop the top of the stack and jump if it is false.
case	Indexed jump.
load_local	Load a value at a given offset from the top of the stack.
indirect	Indirect load using the address on the top of the stack and a given offset.
load_constant	Load an immediate value.
const_addr	Load a value from the constant vector.
call_closure	Call a procedure using the closure on the stack.
call_sl	Call a procedure using a static link.
return	Return from a procedure, popping a number of arguments.
get_store	Allocate a new object of a given size.
move_to_vec	Store a value in a new object.
lock	Lock a new object when all the values have been put in.
set_handler	Set up an exception handler.
del_handler	Delete the exception handler.
raise_ex	Raise an exception.
load_ex	Load the last exception name.
reset	Remove values from the stack.
reset_r	Remove values from the stack, sliding the top value down.
set_stack_val	Change a value on the stack (only used in tail-recursion).
pad	A no-operation.
enter_int	Enter interpreter (only used during bootstrapping).

Table 1: Primitive Opcodes

additions, and whether it justifies the extra complexity in the code-generator.

We will now look at the instructions in more detail and describe some of the derived instructions, which though not primitive, make the machine reasonably efficient. The choices were made on the basis of compiling a large program and counting the occurences of various combinations of opcodes and operands. The most common choices were made into single byte-codes.

3.1 Control and Procedure Call

jump, jump_false, case, call_closure, call_sl, return, pad, enter_int
The conditional and unconditional branch instructions, *jump* and *jump_false,* are generated from high-level *if* and *while* expressions and from tail-recursion removal. The most common cases are short jumps over only a few instructions so branches would seem to be an ideal candidate for using short operands. However it is difficult for the code-generator, which is generating a stream of instructions, to recognise whether a shorter form of the branch can be used until the destination is reached. This would require the code-generator to generate a long branch initially and then to shorten the branch and shift up the code if a shorter branch could be used. Shifting the code is generally a bad idea since other addresses must be adjusted, so instead two forms of each instruction are used, a direct and an indirect jump, each with a single byte operand. The direct form is used for branches of less than 256 bytes, and the indirect form for longer jumps. The code-generator keeps a list of all branches whose destinations are not yet known and compiles in an indirection table for branches which are about to go out of range.

The *case* instruction does an indexed branch to one of the addresses which follow the instruction. This instruction could be replaced by a series of comparisons and conditional branches, but when the number of tests is large it provides a considerable speed-up.

There are two kinds of procedure call mechanisms used. The general case is where a procedure is a closure, in other words the code of the procedure together with the non-local values it uses. It must be used wherever a procedure is returned from a procedure or assigned to a variable.

The representation of a closure is a vector whose first word is the address of the code and whose other entries are the non-local values. The *call_closure* instruction pushes the return address on the stack, then the address of the closure, and then jumps to the code.

Constructing a closure involves allocating space on the heap and copying the non-local values into the closure. This is expensive so where possible an alternative procedure call mechanism is used. If a procedure is only ever called and is not used

in any other way, such as being assigned to a variable, the non-local values it uses will always be on the stack when it is called and they can be found with a static-chain. These procedures are called with the *call_sl* instruction. The operands to the instruction are the procedure address, always a constant, and the static link entry to be used, either the address of the current frame or of another frame found from the static chain if the procedure is not local. Because there is no frame pointer register, in either case the base of the frame has to be found from the stack pointer. The instruction includes an operand which gives the current number of items on the stack so that the base of the frame can be found.

The *return* instruction is used to return from either kind of procedure. The value on the top of the stack is returned as the result of the call. It has an operand which is the number of arguments to be removed from the stack.

The *pad* instruction is no-operation and is used before a call instruction to ensure that the return address pushed onto the stack is not aligned on a word boundary. This is necessary so that the garbage-collector can distinguish return addresses, which can point into objects, from other addresses. While an interpreter could, for example, add a bit to a return address before pushing it on the stack, and mask it out when returning, this would be difficult with compiled machine-code where the return address would normally be a simple machine address. This way of distinguishing return addresses works equally with compiled machine-code.

The final instruction in this group is *enter_int*.
It is only used during the boot-strap phase when a machine-code version of the system must coexist with the interpreted code. Its purpose will be described in due course.

3.2 Data

load_local, indirect, load_constant, const_addr, set_stack_val, reset, reset_r, get_store, move_to_vec, lock

load_local copies a value from the current stack frame onto the top of the stack. It is used to load the values of local declarations or arguments. Because there is no frame pointer the operand gives the number of words from the stack pointer before the operation. Loading from the stack is a very common operation so single byte operations are used to load the first few words on the stack.

The *indirect* instruction replaces the top item on the stack by a value from the object it points to. An operand gives the offset from the start of the object of the value required. Small offsets are very common so single byte operations are used for them.

Often a value is loaded from the closure of a procedure or by following a static chain. The static chain or the closure address are both on the stack so they can be loaded by *load_local*. Loading a value from the closure or following the static chain can be done with one or more *indirect* instructions. Because of the frequency of these operations derived instructions are used for loading closure or static link values.

There are two kinds of value in the Poly system. *Addresses* are values greater than 65536 and are used to refer to objects. They will be changed by the garbage-collector if an object has to be moved to compact the store. *Numbers* have values of less than 65536 and are used for small integers. Either kind of constant can appear in a piece of code but they must be treated differently. Addresses are put in a section at the end of the code where the garbage-collector can find and modify them. They are loaded by the *const_addr* instruction which takes as its operand the offset from the current program counter of the value required. Numbers on the other hand always have the same value so they can be put in the code directly. They can be loaded by the *load_constant* which is followed by the lower 16-bits of the value. Single byte instructions are provided to load common constant values such as zero.

The Poly system is basically a functional machine with operations on variables done by special procedure calls. However tail-recursion removal means that values on the stack can be overwritten when a procedure tail-recurses. In the case of a normal procedure one or more of the arguments will be overwritten with a new value. The *set_stack_val* instruction takes a new value for an argument off the stack and sets the given location to that value. Because inline procedures as well as normal procedures can be tail-recursive it is possible for local values to be assigned new values with *set_stack_val*.

The *reset* instruction removes items from the stack by resetting the stack pointer. *reset_r* does a similar job but slides the value on the top of the stack down.

New objects are allocated on the heap by the *get_store* instruction. This returns the address of an object of specified size. Values are put into the object with the *move_to_vec* instruction which takes a word off the stack and puts it into a given offset in the new object. When each value has been filled in the *lock* instruction is used to indicate that the object is no longer changeable. It is needed because the garbage-collector treats objects that may be changed from objects that may not. This sequence is used for the general case where several objects may be constructed, such as closures for mutually recursive procedures. In most cases a single *tuple* instruction can be used which allocates an object, pops values from the stack into it, and locks it.

3.3 Exceptions

set_handler, del_handler, raise_ex, load_ex
An exception handler is set up by the *set_handler* instruction. This pushes two words onto the stack containing the address of the handler and the previous value of the handler register, *hr*. The register is then loaded from the stack pointer so that it points to these words. The address of the handler points into the code in the same way as a return address so it must not be aligned onto a word boundary. A *pad* instruction may be needed before it to ensure this.

An exception can be raised with a *raise_ex* instruction. The value on the top of the stack is the name of the exception and this is loaded into the temporary register *t*. The stack pointer is then reset to the value in the handler register. The two values on the stack are now the previous value of the handler register, which is reloaded, and the address of the handler code to which a jump is made. In this way the handler is entered with the stack pointer at a known value. The exception name can be recovered from *t* by the *load_ex* instruction.

At the end of the scope of the handler, if an exception has not been raised, the previous handler is reinstated with the *del_handler* instruction. This pops the two values off the stack, loading the handler register with the old value and discarding the handler address. The *del_handler* instruction has an operand which is the location to jump to. This is used because the code for the handler itself is compiled immediately afterwards so a jump must be made round it.

4 Machine-Code Versions

The machine-code code-generators for the Vax and the 68020 are basically very similar to the byte-code version. The original version for the Vax essentially macro-expanded the byte-code instructions, making use of the store-to-store instructions available on the Vax. In due course a more efficient version was written which made better use of the registers.

4.1 Pseudo-Stack

The present version of the code-generators attempts to use registers rather than the hardware stack for local declarations and intermediate results. In the first version of the code-generator a local declaration was generated by evaluating the expression and pushing the result onto the stack. Any references to the declared identifier involved loading the value from that stack location. It is often the case that a local declaration is made and the value used soon after, much in the way

that a temporary result is used. The result of evaluating an expression usually ends up in a register and so if the identifier being declared is used soon after, the value can be taken directly from the register. If there are only a few uses of the identifier it may be possible to use the register for all the references in which case there is no point in pushing it on the stack at all.

Instead of using the real stack for these values a *pseudo-stack* is used in the code-generator to record the location of a value. Entries on the pseudo-stack are either literal constants, values in registers, or addresses relative to a register. Values on the real stack are represented by offsets from a notional frame pointer register and converted into offsets from the stack pointer when the instruction is generated. Making a local declaration involves evaluating the expression and pushing the location of the result onto the pseudo-stack. References to the values on the pseudo-stack use the appropriate location. Values in registers need not be transferred to store unless the register is required for something else, or if a branch or a procedure call is being made which could change the contents of the registers. In that case the registers are pushed onto the real stack and the pseudo-stack is updated to give the new locations.

For each entry there is a use-count giving the number of references to the value. This is calculated by the higher-level parts of the compiler and allows items to be removed from the pseudo-stack when they are no longer required. Registers holding these values can then be re-used. It is also useful to know when the last reference is being made, particularly if a value is in a register. Many operations involve changing the value in a register, for example the addition operation often involves adding a value to the contents of a register. Since this changes the register contents it can only be done if the value will no longer be required.

A highly optimising code-generator might attempt to merge multiple occurences of the same piece of code. On the whole this should probably be done by the machine-independent optimiser, but the code-generator does attempt to remove repeated loads from the same location. The most obvious case is where a value on the stack is loaded into a register by two neighbouring instructions. When an entry on the pseudo-stack is loaded into a register the entry is marked with the register number. If it is to be loaded a second time the register can be used without having to be reloaded. Since values on the stack are simply held as addresses relative to a register this method can be used for any value which is relative to a register. For instance when following the static chain a register is loaded with the head of the chain and then the next frame is found by an indirection off it. This may be repeated until the required frame is found. The code-generator does not remove items from the pseudo-stack immediately if they have been loaded into a register. Instead it keeps them in case they are needed again. When a relative address is put on the pseudo-stack there is a search to see if it is there already and if it is, a reference is made to the existing entry rather than pushing a new entry. If

the existing entry has been marked as having been loaded into a register so much the better and that register can be used if a further indirection is made. In this way following a chain of indirections for a second time will not generate any new code, provided that none of the registers pointing to entries in the chain have been reloaded with anything else. If the chain has been broken anywhere by using the register for something else the chain will have to be reloaded from the break onwards even if further entries in the chain are actually in registers.

4.2 Labels and Jumps

The code-tree represents flow of control by the structure of the tree but this has to be flattened using jumps when the machine code is generated. The canonical example is the if-expression but jumps are also produced from while-loops, exception handlers and tail-recursive calls. Conditional branches in the VAX instruction set only allow offsets of $+127$ or -128 bytes which is insufficient to branch round more than a few instructions. There is an unconditional branch instruction which can be used together with the conditional branch to make longer jumps. A long conditional jump can be made simply by using a conditional branch round an unconditional long branch and having the condition inverted, but this would be a considerable overhead if used for every conditional branch. It is better to try to use the short conditional branches where possible since many, perhaps most, of the branches will be within the range of a short branch.

As with many problems there is a trade-off between the improvement in the code gained by using short branches and the effort involved, but it is possible to go a long way without a great deal of effort. To solve the problem completely requires multiple passes because the size of one branch instruction may affect whether another can use the shorter form [4]. The solution adopted in the present version of the code-generator uses only the single pass but generates reasonably good code. It is based on a scheme described in [1] and assumes that long branches are only needed rarely. Short branches are used for all forward jumps, which works fine provided the code between the jump and its destination is not too large. We may, however, put in a short jump and start generating the code which it is jumping round, only to find that the short jump is not going to reach. The solution is to interrupt the code that is being generated and put in a long unconditional branch to carry on the jump. The destination of the original short jump is set to this new branch instruction and this long branch should now be enough to reach round any more code. Of course this new branch instruction cannot be put in anywhere we find a branch coming to the end of its range. We must ensure that the normal instruction stream is unaffected by having this long branch put down in the middle of it. This is done by only putting the long branch between two instructions and preceeding it by an instruction to jump round it.

```
        tstl r0 # test a value
        beql L1a# conditional branch, originally to L1
        .
        .   A lot of code
        .
        brb SKIP# skip the branch extension
L1a:    brw L1  # continue the conditional branch to L1
SKIP:   .       # and continue the previous code
        .
        .   Perhaps a lot more code
        .
L1:     ...     # The final destination
```

There is a fixed overhead in generating this code because of the need to branch round the extension. To reduce this there are two limits which are used to decide whether to extend a branch. The *hard limit* is reached when an instruction is about to be generated which will almost certainly cause a jump to go out of range. At this point the instruction stream must be interrupted and that jump extended. There may, however, be other jumps which, while they are not at the extreme of their range, are getting close to it. Having once interrupted the instruction stream and generated the skip round the extended jump, it is probably worth extending these other jumps as well. The *soft limit* is used to decide whether jumps are getting close enough to the end of their range to extend them if the stream has already been interrupted. The soft limit also comes into play if an unconditional branch is generated as part of the normal instruction stream since there is then no need to interrupt the stream.

5 Forward References

Recursive and particularly mutually recursive procedures pose a problem for the code-generator since it is not possible to get the address of the code of a procedure until it has been compiled. This means that some procedures may be compiled before all the procedures they refer to. Since there is no separate linking phase forward references have to be filled in by the code-generator.

References

[1] R. D. Evans, *Language Implementation in a Portable Operating System*, Ph. D. Dissertation, University of Cambridge. 1981.

[2] Robin Milner, *The Standard ML Core Language*, Edinburgh University.

[3] David C.J. Matthews, *The Poly Manual*, SIGPLAN Notices, Septemeber 1985.

[4] M. H. Williams, *Long/Short Address Optimization in Assemblers*,Software - Practice and Experience Vol 9,227-235 (1979)

Machine Independent Optimisation in Poly and Poly/ML

David C.J. Matthews

1 Introduction

The Poly and Poly/ML code-generators generate a machine-independent code which is processed by an optimiser before being passed to a machine-dependent code-generator. The purpose of this note is to describe the machine-independent code and the optimisations performed on it.

The machine-independent code is in the form of a tree structure, the code-tree, which is easier to manipulate than the more conventional sequence of instructions, and with a persistent storage system does not have to be flattened to be preserved. Each node of the tree contains an instruction together with some other data and/or pointers to other nodes. For example a node containing a function call, the eval instruction, has a pointer to code returning the function, a list of arguments and the number of arguments.

The code-tree could be directly interpreted by a recursive procedure which walked over it evaluating the expressions and returning the results. At the most basic level the code is purely functional though there are some special instructions for creating and manipulating a "store". Declarations are handled by the "decl" instruction which saves the result of an expression on the "declaration stack". The value can then loaded by the "extract" instruction which gives the location in terms of an offset in one of currently accessible stack frames. As described below, references to non-local declarations must be converted into addresses in the closure of the procedure if higher-order procedures are to work. There are no references to global declarations in the code since they are always replaced by their values when they are first looked up.

2 The Optimiser

The function of the optimiser is to perform transformations on the code-tree which preserve the original meaning, but allow the machine-dependent code-generator to produce better code. A simple example is constant propagation. A declaration may consist of a giving a name to a manifest constant. This will generate code to put the value onto the stack. References to this declaration will be made through the identifier which will generate code to load the value off the stack. If the two are

matched up by the optimiser then the instruction to load the value can be replaced by the value itself, and the instruction to push the value onto the stack can be deleted. A similar process is used where a declaration involves simply giving a name to a value which has been declared previously e.g. let x == ...; let y == x. Declarations of inline procedures are treated in the same way. Simplifying these declarations is particularly important when expanding inline procedures.

A large part of the work of the optimiser is in processing inline procedures. Many instructions in Poly are implemented as inline procedures, for example the instructions which extract fields from records or even the instruction to add two integers together. In addition there are polymorphic functions which select the type from which an operation is to be applied, for example the + function which applies **integer$+** if the arguments have type **integer**. This means that adding two values together involves two inline procedure expansions.

In essence an inline procedure is expanded by replacing the call by a block consisting of declarations of the formal parameters with their actual values, and then the body of the procedure. The process described above for removing redundant declarations is used to simplify the code. A few problems are found with recursion. If the inline procedure recurses then it cannot of course be expanded indefinitely. Instead the recursive call is generated as a normal procedure call. Tail-recursion is treated specially and is passed on to the code-generator to generate a jump back to the start of the body of the inline procedure. This adds a few complications. The start of the body of the procedure, as opposed to the code which declares the actual parameter values, must be marked. The second complication is that any parameters which may change as a result of the recursion must be forced into variables and not optimised out even if their initial values are constants. Optimising tail-recursive inline procedures is probably worthwhile because it allows **for** and **first** functions which sequence over abstract sets to be implemented as efficiently as a built-in **for** statement.

The optimiser detects tail-recursion of ordinary procedures and marks tail-recursive calls so that they can be processed specially by the machine-dependent code-generator.

One function the optimiser performs is not optimisation at all since it must be done to allow higher-order functions, and that is converting references to identifiers outside a procedure into references via the closure. A procedure in Poly is represented as the code of the procedure together with the non-local values referred to by it. It is implemented as a vector whose first word is the code and subsequent words are copies of the non-local values. Since variables (references) are not held directly on the stack all values put in the closure can be copied without affecting the semantics. When a procedure is declared the closure is constructed and that is the value which represents the procedure. It can be passed around or assigned to variables and it will always work. This implementation has the advantage that a stack can be used for all the local values and stack frames do not have to be processed by the general storage allocator.

Creating a closure is still an expensive operation since the vector must be allocated and the values copied. It is frequently the case that a full closure is not needed, as is shown by the number of languages which manage without procedures as first-class values at all. A common case is the procedure with no non-local references at all. This case is handled specially by combining the closure and the code. Another case is where the procedure has non-local references but is used in a way that does not require a closure. If a procedure is only ever called and is never returned as a result from a procedure, passed as a parameter to a procedure (this includes assignment to a variable) or referred to from the closure of another procedure then it need not be made into a closure. The optimiser keeps track of references to a procedure and marks it as not needing a closure if the conditions are satisfied. The code-generator can then use a static link or a display for these cases.

The optimiser processes the declarations of a block by recursing down the block optimising each declaration or statement in turn. As the recursion unwinds it looks at the procedures which have been declared and marks the procedures which do not need full closures. Those that do need full closures have all the non-locals they refer to marked as "loaded" and so they will be made into closures if they are procedures themselves. This works because the scope rules (excluding mutual recursion) ensure that a procedure only refers to declarations made before it, i.e. earlier on the list.

A situation that occurs frequently in ML is generating tuples which are immediately taken apart by other code. The optimiser detects such cases and avoids generating redundant tuples.

Hardware Support for Poly and ML

David C.J. Matthews

So far Poly has been implemented on the Vax and the Sun (MC68020) using native code. There is also a version which generates a byte code that can be interpreted. Even this version, however, was intended as an aid to porting the system from the Vax to the Sun and was designed to be very similar to the native code versions. It would be a very interesting exercise to design an architecture specifically for Poly and ML and to see how would differ from a conventional machine. The rest of this note proposes a possible architecture based on the abstract machine used in the byte-code system.

The architecture was developed from a code-generator for a conventional machine so the instructions are fairly low level. It would be possible to provide instructions which would be closer to the original Poly or ML code, such as an instruction to extract all the values from a tuple onto the stack as a single operation, but there are advantages, particular in the machine-independent optimiser, of having a simple instruction set. It is always open to the designer of the hardware to provide additional instructions which implement a combination of the primitive instructions, and indeed that would probably be necessary to achieve reasonable performance. That should be done using data about the occurences of particular instructions and operands in practice.

The basic machine is intended to be extremely simple, with few registers, no complicated addressing modes and relying heavily on a second processor for many of its operations. This would allow the machine to be developed by stages, with an initial version supporting only the basic instructions, but later versions performing more of the functions of the second processor.

Finally there are certain operations which are possible with specialised hardware which are not possible on a conventional computer, particularly to do with virtual memory and asynchronous garbage collection. Some suggestions along these lines have been made but there are no specific proposals.

1 Overview

The abstract machine can be thought of as two separate processors with access to a common memory. It might be implemented as one or several real processors. The *primary* processor has a small set of primitive instructions to do operations such as jumps, function calls and loads. The *secondary* processor is activated to deal with virtual address faults and to perform various *primitive functions*. It is also responsible for garbage collection. The *primitive functions* are operations that behave like functions in Poly or ML but cannot be written using the instructions of the primary processor. They extend from simple operations like adding together two integers through to complex operations like opening a file. Many of the simpler operations might be handled for efficiency reasons by the primary processor but there are advantages in, conceptually at any rate, dividing the operations up in this way.

2 The Primary Processor

The primary processor is a stack machine with a single address space. Each process has its own stack segment. The stack pointer, *sp*, points to the top entry on the stack which is used for instruction operands, function parameters and results, return addresses and for stacking exceptions. The program counter, *pc*, points to the current instruction in the code. The handler register, *hr*, points into the stack at the current exception entry. The stack base register, *sb*, points to the base of the current stack segment. The temporary register, *t*, is only used during exception handling. Store is allocated from a contiguous heap segment pointed to by the allocation register, *ar*, with the limit of the heap indicated by the heap limit register, *lr*.

The instructions of the primary processor are fairly limited (Table 2).

3 Secondary Processor

The secondary processor is responsible for input/output and handling virtual memory faults. Many of its functions might well be implemented in the same physical processor as the primary processor. Table 3 gives a list of the functions currently implemented in the Poly system.

Apart from *nullorzero* which is a null string, all these entries are functions. The arguments are popped from the stack, and any result returned on the stack. To do this the secondary processor must be able to find the top of the stack and adjust

jump	Unconditional jump.
jump_false	Pop the top of the stack and jump if it is false.
case	Indexed jump.
load_local	Load a value at a given offset from the top of the stack.
indirect	Indirect load using the address on the top of the stack and a given offset.
load_constant	Load an immediate value.
const_addr	Load a value from the constant vector.
call_closure	Call a procedure using the closure on the stack.
call_sl	Call a procedure using a static link.
return	Return from a procedure, popping a number of arguments.
get_store	Allocate a new object of a given size.
move_to_vec	Store a value in a new object.
lock	Lock a new object when all the values have been put in.
set_handler	Set up an exception handler.
del_handler	Delete the exception handler.
raise_ex	Raise an exception.
load_ex	Load the last exception name.
reset	Remove values from the stack.
reset_r	Remove values from the stack, sliding the top value down.
set_stack_val	Change a value on the stack (only used in tail-recursion).
pad	A no-operation.

Table 1: Primitive Opcodes

Short integer arithmetic
mul_int, plus_int, minus_int, div_int, mod_int, neg_int, int_eq,
int_neq, int_geq, int_leq, int_gtr, int_lss, printi, repri, convint

Boolean operations
or_bool, and_bool, not_bool, printb, reprb

File operations
openstream, readstream, writestream, closestream, eofstream, lookahead,
stdin, stdout, popenstream, pclosestream

Process operations
fork_process, choice_process, kill_self, int_process, send_on_channel, receive_on_channel

Arbitrary precision arithmetic
aplus, aminus, amul, adiv, amod, aneg, testa, printl, lconvint, reprl

Real number operations
Add_real, Sub_real, Mul_real, Div_real, Comp_real, Neg_real, Print_real, Repr_real,
conv_real, real_to_int, int_to_real, sqrt_real, sin_real, cos_real, arctan_real, exp_real, ln_real

Character operations
succ_char, pred_char, convch

String operations
string_length, prints, strconcat, string_sub, substring, teststreq,
teststrneq, teststrgtr, teststrlss, teststrgeq, teststrleq

Word operations
mul_word, plus_word, minus_word, div_word, mod_word, neg_word, or_word,
and_word, xor_word, shift_right_word, shift_left_word, word_neq,
word_geq, word_leq, word_gtr, word_lss, word_eq, printw, reprw, wconvint

Indexed loads and stores
load_byte, load_word, assign_byte, assign_word

Other operations
exit, install_root, chdir, sysexec, callcode, filedate, gettime, filemode,
ptime, repr_time, io_operation, commit, createf, getstore, lockseg,profiler

The null string
nullorzero

Table 2: Primitive Functions

the stack pointer. The interface between the two processors involves the primary processor saving the registers in the base of the stack segment and passing the contents of the stack base register, *sb*, with the function required, to the secondary processor. The secondary processor can use this to find the arguments on the stack, adjust the registers, and return control to the primary processor. The result it returns is the stack base register, allowing the secondary processor to change the process being run by the primary processor.

As well as a request to the secondary processor to perform a specific function the primary processor may also transfer control to it in a number of other cases. If the stack segment for the current process is about to overflow, if the heap has run out, if a virtual memory trap occurs or if the secondary processor interrupts the primary processor, the secondary processor can call to the primary processor. These special cases can be represented by special function values.

The great advantage of separating the functionality between the primary and secondary processors is flexibility. It would seem to be convenient to have a separate physical processor to deal with input/output and it is sensible to define the interface between the processors at a reasonably high level. This processor could be a conventional computer possibly running a standard operating system and regarding the Poly primary processor as a peripheral. Once a device driver has been written to allow the secondary processor to read and write to the memory of the primary processor, to interrupt it and accept requests, it is then relatively easy to add functions to Poly or ML running in the primary processor. It might, for example, be convenient for the first implementation of the primary processor to be a very simple machine supporting only the primitive instruction set and passing all the primitive functions to the secondary processor. Further implementations might support some of the functions, such as adding short integer values, within the primary processor. This would be completely transparent as far rest of the system is concerned. Equally, new primitive functions can be provided, to drive a window system for example, simply by writing suitable code in the secondary processor and using an escape mechanism in Poly or ML to associate a particular function number with a function of a particular type. Since the primary processor will not recognise this function number it will be passed to the secondary processor to be executed there.

4 Specialised Hardware Operations

Implementing a language on special hardware can allow for operations which would be too expensive on a conventional machine. There are two specific areas which would be worth pursuing.

The present Poly and ML system makes use of a persistent storage system to allow

for long-term storage of data. Objects are read in from disc as required, and then modified objects can be written out. Because the type systems of Poly and ML restrict the kinds of objects that can be modified to references, arrays and their analogues, it is possible to make this quite efficient. Hardware support for this could improve performance still more.

The other area in which hardware support would be useful would be to provide for asynchronous garbage collection. There are various methods for doing this but they all require too large an overhead on conventional machines. This is basically because they require special action when an assignment is made while the garbage collector is active. With special hardware, however, it would be possible to arrange for the hardware to execute different versions of the indexed store functions depending on the state of some register.

The Poly and ML System: Low Level Support

David C.J. Matthews

9 December 1987

1 Introduction

This document forms part of a set describing the Poly[2] and Standard ML[1] systems. It is concerned with the low-level part of the system and the interface to the outside world. In particular it describes the data structures used to represent various high-level objects, and the algorithms used in the garbage collector and other low-level routines. It is intended primarily for those interested in the details of the implementation or planning to implement their own compiler for Poly, ML or a similar language.

2 Underlying Representations

The lowest level objects handled by the abstract machine are words and vectors. Before describing how the higher level objects such as strings and integers map into them the format of these objects will be described.

A word is a 32-bit value which is either a number or the address of a vector, distinguished by all numbers being less than 65536. Addresses may be either local store addresses or *persistent addresses*. Persistent addresses arise from the persistent storage system which is a kind of "demand-paging" of objects from a database into main store. The persistent storage system is described in a separate report[3] and as far as the compiler is concerned all addresses can be treated as local addresses, with transfers from backing store being done transparently. The only places where the persistent storage system has to be taken notice of is where a persistent store trap might cause a garbage-collection to take place. This requires rather more care in the code-generator than in a system where the garbage-collector can only be invoked when the storage allocator is explicitly called. It also means that testing for address equality, in ML this can happen when two reference values are compared, has to take account of the case of one address being a local store address and the other the persistent address of the same object.

Figure 1 shows the general form of a vector. The first word is the length of the segment in words together with a flag in the top byte. For a normal vector this flag byte is zero. Table 2 gives the meaning of the bits in the flag byte.

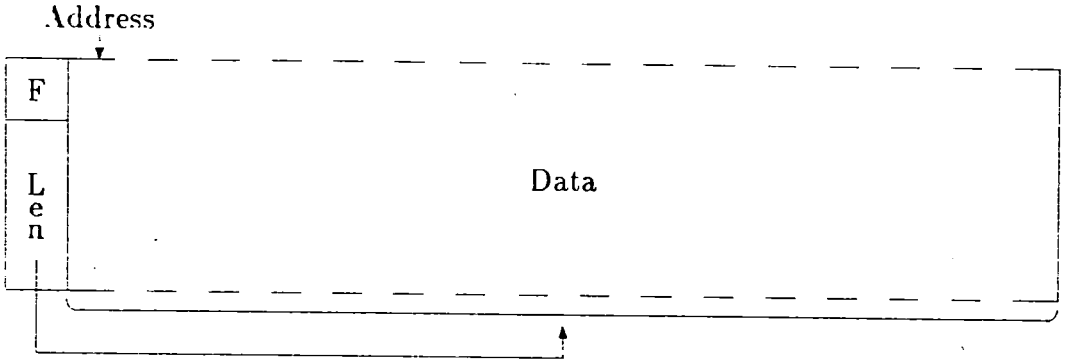

Figure 1: Vector

Bit	
0	Byte segment.
1	Code segment.
2	First word points to second.
3	Stack segment.
4	Application dependent.
5	Application dependent.
6	Segment is "mutable".
7	Used by the garbage-collector.

Table 1: Flag bit definitions

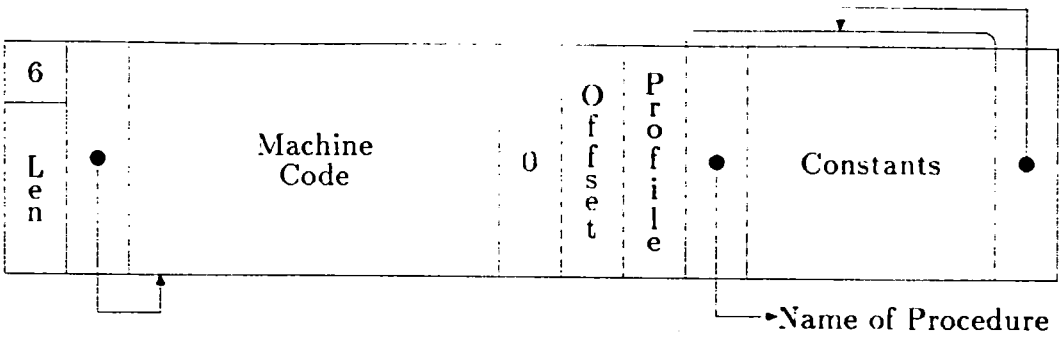

Figure 2: Format of a Code Segment.

Apart from the **application dependent** bits, one of which is used as a sign bit for arbitrary precision integers, the bits are only used by the store-management system. The **mutable** bit is set for segments such as variables (references) or arrays where the contents can be changed by assignment. It can be set when a segment is created and switched off later, for example to construct circular structures, but an immutable segment cannot be made mutable. It is used by both the persistent storage system and the garbage-collector where mutable and immutable segments are treated differently.

The **byte-segment** bit is used where a segment contains values which cannot be addresses of other objects. If this bit is not set the garbage-collector assumes that each word is an address of an object if it has a value greater than 65535. It is essential that the characters in a string, for instance, are not treated as though they formed addresses. **Byte-segments** are also used for long integers and reals and for some segments of executable code.

A **code-segment** is a combination of an ordinary segment and a byte-segment and is used only for segments of executable code which contain constants. Constants are either literal values, or arise from references to top-level declarations. It would be possible to load them into the closure of the procedure which uses them but it is better to reduce the size of the closure by binding them directly into the code. They may be addresses so they must be inserted into the code in such a way that the garbage-collector can find them. The first part of a code-segment contains the code and this is treated by the garbage-collector as a byte segment. The constants are put at the end of the segment and the last word is the number of constants.

Figure 2 shows the format of the most general code segment which is a code-segment with bit 2 set. Like all vectors the first word contains the length field with the top byte used for a format code, in this case 6, and the address of the segment is the first real word of the segment. The end of a code segment has a special format which is used by the garbage-collector and other procedures. After the code there is a single word containing zero. The code-generator does not generate whole words with zero so this word acts as a marker. After the marker there is a

word which contains the offset from the start of the vector. When the garbage-collector finds an address which points into the code, either a return address or the address of an exception handler, it must be able to locate the start of the vector. It searches for the marker and then subtracts the value of the next word from the address to get to the start of the vector.

The next word after the offset is used for profile counts. It is normally zero but if profiling is on this word contains the count. The profiler finds the word in the same way as the garbage-collector.

After that comes the name of the procedure. This is a string so it will normally be an address unless the name happens to be only one character long. It is used by the profiler when the profiling counts are printed out and by the tracing code when a trace is requested *via* a console interrupt.

Following the procedure name come any constants used by the code and the final word is the number of constants, including the procedure name. When the garbage-collector is tracing a format 2 or format 6 block it only looks in the constant segment for possible addresses and ignores the remainder of the vector. Of course when a format 6 block is moved during compaction the first word must be updated to the new location.

The **first-word-points-to-second** bit is only used with code-segments or byte segments containing code. This bit is set in a segment which contains the code of a procedure which has no non-local references, other than to constants. When a procedure is declared it is generally necessary to construct a closure which is represented by a vector. The first word contains the address of the code and the remaining words contain the non-local values used by the procedure. A procedure is called by loading the address of the closure into a register and doing a subroutine call indirecting through the first word. A procedure with no non-local references would have a closure which consisted of one word which was the address of the code. To save having odd one-word objects the closure and the code are combined into a single segment whose first word, instead of being the start of the code is the address of the seond word which is the real start of the code. Such a segment has bit 2 set and a separate closure is not created. The calling sequence however is exactly the same as if it had a closure so this segment can be passed around as though it were a closure.

The stack for each process is held in a **stack-segment** whose format is shown in figure 3. The format is designed to be usable on different machines with different numbers and arrangements of registers. The store-management system assumes that all the saved registers contain values which are either less than 65536 or point to the start of a vector. This allows the garbage-collector to treat them as ordinary addresses and to change the value if it moves a vector that a register points to. Three saved values are used for special purposes and are saved in specific fields. The stack pointer field, **sp**, contains the lowest extent of the stack, which grows down from the end of the segment. The program counter field, **pc**, points into the code which was being executed when the process was suspended, and the the

handler field, **hr** points into the stack at the last exception handler data. These fields are used for producing stack traces and profiling information.

Normally values are either numbers or point to the start of vectors. However values on the stack are allowed to break the rules in two ways. The stack may contain pointers elsewhere on the stack, for instance for static links and for linking exception handlers. In addition it may contain pointers to the middle of pieces of code, either as return addresses or as exception handlers. To allow these to be distinguished from other addresses, addresses into code are never aligned on word boundaries. This is enforced by the code-generator.

3 Higher Level Values

All higher level objects are implemented in terms of these basic low level objects. This section describes how the basic types are implemented.

3.1 Structured Objects

Records and structures in Poly or tuples in ML are implemented as vectors of words. Unions are implemented as pairs with the first word containing a tag, and the second containing the injected value.

3.2 Integer and Long_integer

There are two representations of integers in Poly. The type `integer` is a short precision value between -32768 and 32767. It is represented as a word containing the number itself in the lower half of the word, with the top half zero. In Poly there is also the arbitrary precision integer type `long_integer` which is used for `int` in ML. The representation of numbers between -32768 and 32767 are the same as for `integer` making conversion easy and allowing operations on small values to be done rapidly. Numbers outside this range are represented by the address of a vector of bytes. A sign-and-magnitude representation is used with the sign-bit in the flag bits of the vector. Each byte represents a base-256 digit and the vector represents the number, low-order byte first.

Operations on short precision numbers are compiled into the code directly using the normal machine-code integer operations. However, the arbitrary precision package is written in C so operations on arbitrary precision numbers have to be done by procedure calls. In the Sun and Vax implementations the interface procedures which call the C from Poly or ML and are written in assembly-code, handle the special case of an operation where the arguments are small numbers and the result is small.

3.3 Real

Neither Poly nor ML are intended for applications which make intensive use of floating point numbers so the implementation is not particularly efficient. The representation used for real numbers is a double precision value held in a byte-vector. All operations have to be done by procedure calls.

3.4 String and Char

Strings are normally represented by the address of a vector of bytes. The first word is a number which is the number of characters in the rest of the string. Having a 32-bit length count means that the length of strings is constrained by storage availability rather than by any inherent limit.

Null strings are represented by a string with a zero length field. For efficiency a word in the interface vector is used for this and all the string operations return a pointer to this if they need to return a null string. Single character strings, including the string containing the ASCII NUL character, are represented by the character value itself. This makes some operations rather more complicated but saves a considerable amount of store, since otherwise each single character would be represented by two words. This means that a single character string has exactly the same representation as a character.

3.5 Streams

Input and output streams work through the normal Unix standard I/O mechanism however the Poly program does not operate directly on the streams. Instead the value returned by opening a stream is the address of a token which contains an index into the **stream table** which is part of the run-time system data structures outside the Poly heap. Each entry in the table is a pair of values, one of which is the address of the token and the other the address of the Unix stream. A new token is allocated on the heap when a stream is opened (figure 4), and when a stream is closed the token address in the entry is cleared (figure 5). Every time the read or write functions are called the run-time system checks that the token address part of the entry corresponds to the token that was provided. If it fails to match the function raises an exception. This allows an entry in the table to be reused if a new stream is opened but in such a way that a previous stream that has been closed will not appear to have been reopened. In figure 6 S1 has been closed and S2 reopened using the same entry. Because the token address in the entry does not point to S1 any operations using S1 will raise exceptions.

This is also useful if the database has been written out with streams open. The stream tokens will be written out and if they are read in to a new session will fail to match. The exception is the standard input and output streams, whose tokens are entries in the interface map. They are linked up by the persistent storage system so they will always match.

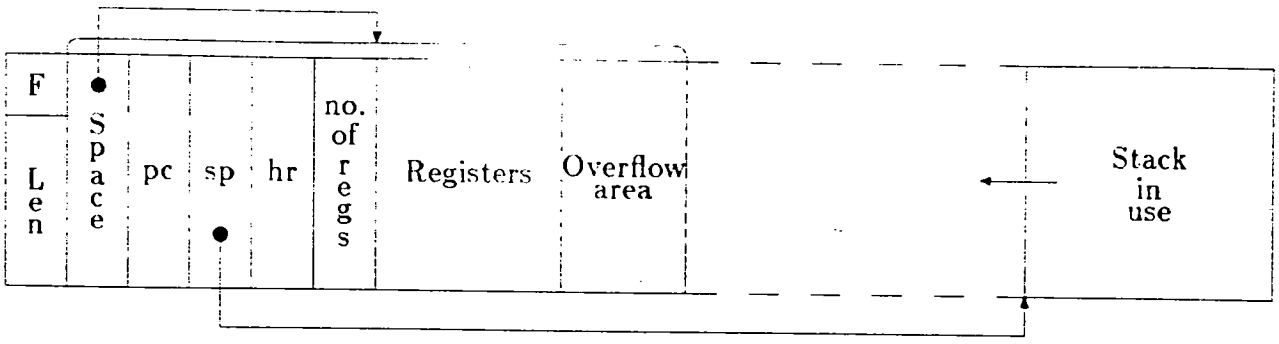

Figure 3: Format of a Stack Segment.

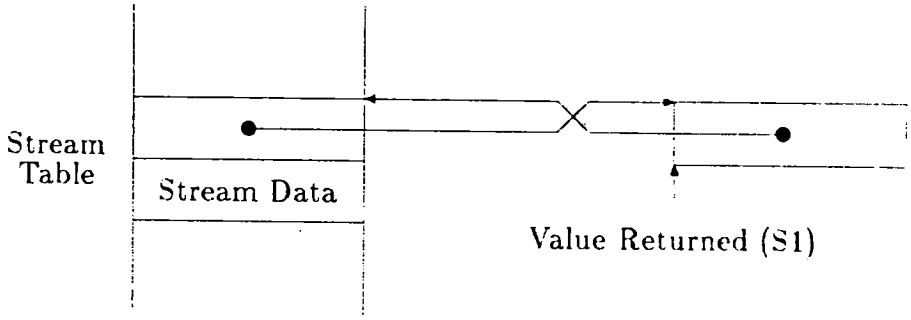

Figure 4: After Opening a Stream.

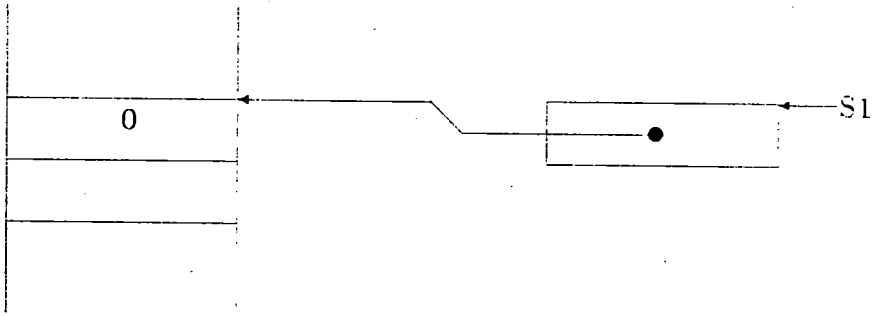

Figure 5: After Closing the Stream.

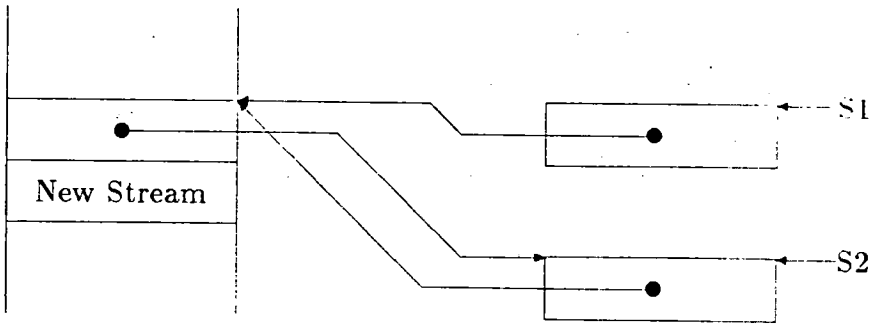

Figure 6: After Opening another Stream, Reusing the Entry.

Another use for this is to try to ensure that unreferenced streams are closed automatically. The stream tokens are held on the heap and may be moved by the garbage-collector if the heap is compacted. The corresponding addresses in the stream table must be updated so that they will still match the tokens. It is possible that a stream may be become no longer referenced before it has been closed. In that case there can be no further activity on it and in particular it will not be explicitly closed. We can detect that the stream has become unreferenced by the garbage-collector finding that there are no references to the token, apart from the entry in the stream table. In that case the stream handler closes the stream. All this is really only necessary if there is a limit on the number of streams that can be open at once.

A similar mechanism has been used for a window system. There, however, an attempt to do an operation on a window whose token does not match can cause the window to be reopened. The token is more complicated than for a stream and holds information about the window's size and position. This is used to reactivate windows when a new session is started, using information from the persistent store. If a window has been explicitly deleted that is marked in the token, and operations on it will raise an exception.

4 Processes

The process mechanism in Poly is mostly implemented in the run-time system. Each process has a process block and a stack which is initially quite small, but it can grow if necessary by allocating a new stack segment and copying the data over into it. A register points to the lower end of the stack so that a simple comparison with the stack pointer will normally suffice to check for overflow. A larger stack can then be made, unless the size has become too large when it is assumed that the process is in an infinite recursion and an exception is raised.

Stack overflow checking also serves another purpose. It is sometimes necessary to interrupt a process when, for example a process has exceeded its time-slice. However a few pieces of code have to be treated as single units, a particularly common example being the instructions which allocate store off the heap. So, instead of interrupting the code immediately, the pointer to the end of the stack is changed so that the next time the code checks for stack overflow it will trap. The code that would normally handle the overflow trap can recognise this case and call the process scheduler.

The processes communicate on blocking channels. A channel is represented by a pair of words one of which is used as the head of a chain of processes waiting to send on the channel and the other the chain of processes waiting to receive. It is possible, but unusual, for both the chains to have processes in them.

Processes are created by two different run-time system calls. The simpler is **fork**. This creates a process that runs in parallel with its caller. It takes two arguments, one which is a function to run as the process, and the other which

is a switch which determines whether the process is to receive console interrupts as **interrupt** exceptions. Runnable processes are put into a doubly-linked list. Changing the current process simply involves changing the pointer into this chain. When a process wants to send or receive on a channel and there is no process waiting to do a corresponding transfer the process is blocked. It is taken out of the runnable chain and added to the appropriate list for the channel. Note that it is possible for such a process to be garbage-collected if the channel it is waiting for is no longer reachable from a runnable process.

The other way of creating a process is with **choice**. This takes two functions as arguments and runs them as processes. Unlike processes created with **fork** these *choice* processes are mutually exclusive as far as sending and receiving messages are concerned. If one successfully sends or receives a message the other will not be allowed to. The interlocking between the processes is done using **synchroniser** values.

A process can also be blocked if it attempts to read from a stream when there is no data available. The process is suspended in such a way that when it is restarted it will try reading again, either succeeding if data is now available, or being suspended again. There is a timer which interrupts the current process and starts another one. This ensures that processes get a reasonably fair share of the machine and that processes waiting for input will run when the input arrives.

5 Addresses and the Persistent Store

The persistent storage system is designed to be transparent as far as the machine-code is concerned. The code can operate on addresses without needing to know whether it is using real or persistent addresses and the persistent store handler will take care of this. There are however two places where the use of the persistent store affects the code-generator, in comparing addresses and in the effect of garbage-collection.

Equality of addresses is used for equality of structures in Poly and for equality of references in ML. In most cases this could be done with a simple compare instruction. However it is possible to have a persistent address and a local address which refer to the same object. These should clearly be considered as equal even though the addresses are represented in different ways. To get this right address comparison is done by code which calls a routine in the persistent storage system if it finds two addresses one of which is a persistent address and the other local.

The other point about the persistent store is that garbage-collections can happen at many more places than in a system without a persistent store, where garbage-collections only take place when the store-allocator is called explicitly. This is important because the garbage-collector must be able to find and update all the addresses of objects currently in use which means that the state of all registers in particular must be well-defined. The code-generator must be aware of the potential garbage-collections and ensure that the state is well-defined at each

of them. Between these places it can use the registers as it likes. The garbage-collector for the Poly/ML system assumes that every word in a register or on the stack is the address of an object if its value lies in the range of possible heap addresses. This requires all addresses to point to the start of an object and not inside one. Normally this is quite easy to achieve but a few cases have to be treated specially, usually by clearing a register after it has held a value which would break the rule.

6 Linkage Conventions and Register Usage

A similar calling sequence is used in each of the versions of the system. This has advantages because the format of a stack frame is then the same and so code in the garbage-collector and for producing trace information can be independent of the machine.

The standard calling conventions used on the VAX and MC68020 are not suitable for Poly for a number of reasons. On the VAX this is because the standard call instruction is too slow, and on the 68020 because it is possible to make use of a special return instruction which is not normally used.

The conventions differ slightly according to whether the function is being called with a static link or as a full closure. In the case of static link call a register contains the static link and the subroutine jump is to the start of the code. For a closure call the register contains the address of the closure and an indirect jump is made using the address in the first word of the closure as the destination. The arguments to the function are passed on the stack so after the call the stack contains the return address, pushed by the subroutine call, and the arguments. No other values are explicitly put on the stack. The register with either the static link or the closure address is marked on the pseudo-stack and will be pushed onto the real stack if the register is required, in particular for another function call, in same way as any other register. Small functions which do not do any function calls themselves, never need to push this register and any references to variables in the closure can be done by indirecting through the register.

The code-generator keeps a note of the number of items on the stack and addresses all the values on it using the stack pointer register. This avoids the need for a frame pointer register but makes reading the assembly code listing somewhat difficult because the offsets of local values and arguments change according to the number of items currently on the stack.

6.1 Exceptions

Raising an exception is similar to calling a procedure except that instead of returning to the point of call execution continues with the code following the handler. Exceptions are set up using a *handler register* which is either a real machine register, as on the 68020, or a location in store, as on the VAX. When a handler is set

up a number of words are pushed onto the stack. The first is current value of the handler register, and then follow a number of pairs of words. Each pair consists of an exception identifier and the address of the code to handle the exception. The handler register is then set to the value of the stack pointer, in other words to the address of these words. In this way exceptions are chained together.

If an exception is raised the chain is searched for a handler with an identifier which matches the identifier of the exception. When a match is found the stack is reset to the handler, the old handler register value is reinstated and the handler code is entered. If control reaches the end of the scope of an exception handler without the handler itself being executed the previous handler is reinstated and the handler address is removed from the stack.

Zero is a reserved exception identifier and is used for a handler which will catch all exceptions. Zero as the address of the handler code is also reserved and is used by the exception tracing mechanism. The **exception_trace** function pushes a special handler on the stack with exception identifier and handler addresses both zero. It then calls its argument, a function. If an exception is raised inside that function or somewhere called from it the normal exception mechanism will involve a search for a handler. If there is no handler inside the function the special handler will match. Instead of jumping to the code, however, the exception mechanism prints a stack trace. It then reraises the exception.

7 Storage Allocation

Storage allocation is a major problem when trying to implement languages like Poly and ML efficiently. There are several competing requirements which complicate the problem.

Store may be needed for a number of purposes. Poly or ML code needs space for local declarations and for objects such as tuples or closures. The persistent storage system needs store for blocks that are brought into store and for storage maps. The input/output system must create buffers in store and some other system calls need workspace. An ideal situation would be to have several different regions in virtual memory each able to expand independently so that each requirement could be dealt with in a separate region. Unfortunately the Unix operating system allows there only to be two segments, a *stack* segment that expands automatically as space is required, and a *heap* segment which can be extended by calls to the operating system.

The names of the segments reflect the way that C programs make use of them but there is no particular reason why Poly and ML should follow that usage. It is perfectly possible, for example, to use the heap area for the stack. Indeed if a multi-process system were implemented for Poly the stacks would probably be allocated there. In practice it is convenient to follow the C usage to simplify calls from Poly/ML to the run-time system written in C. In the interpreted system the stack is a vector allocated in the C stack, but in the Vax and 68020 versions the

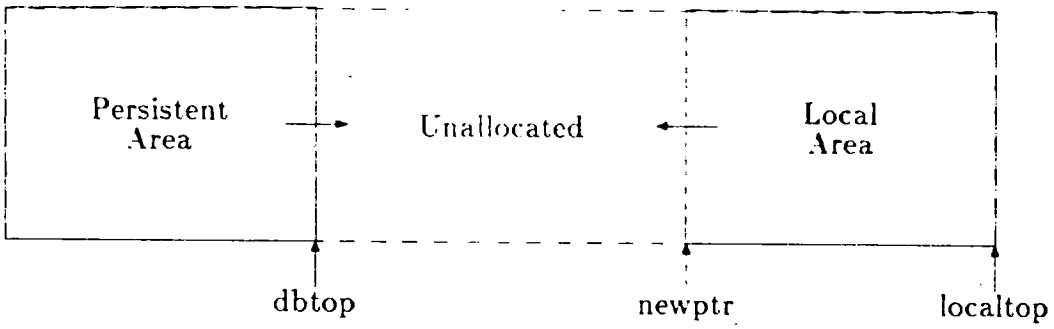

Figure 7: Overall Heap Layout

stack is shared between Poly and C with special pieces of assembly code handling the conversion between Poly and C calling conventions.

Dividing up the heap segment is more difficult. After some experimentation the present system was arrived at where the heap is divided in two between the *persistent* area at the bottom of the heap growing upwards, and the *local* area at the top growing downwards 7. When the areas meet the garbage-collector is called, but only the *local* area is collected.

The *persistent* area contains data read from the persistent storage system and objects in it are arranged as blocks of a fixed size. Rather than garbage-collecting this area it would be better to write the blocks back to disc if they were not being actively used, but in practice this has not been found necessary.

This area is also used for buffers and other objects created by operating system calls. If these were allocated in an area which could be compacted by the garbage-collector their internal structure would have to be known so that addresses in them could be traced. These objects are managed by replacing the standard storage manager calls *malloc* and *free* by versions which maintain a free list of areas among the persistent blocks.

The *local* area is used for objects created by the Poly or ML code or for strings or arbitrary precision numbers created by the run-time system. Allocating store for tuples or closures is a frequent operation and must be made efficient. If micro-code support is available it is possible to have quite complex allocators without reducing the speed significantly, but if allocation is done by macro-instructions then the only efficient method is to have a contiguous area of memory. The allocator then merely subtracts the size required from the current heap pointer, compares with a limit and takes a trap if it has overflowed. This can be done in very few instructions if a trap instruction rather than a subroutine call is used to enter the garbage-collector. On the 68020, for example, this is three instructions and can be compiled in-line rather than using a call to an allocator.

8 Garbage Collector

In a language like ML objects such as lists or closures are created and may then be no longer required. For the system to be efficient the store occupied by them must be recovered so it can be reused. This requires garbage-collection to sort out the objects which are still needed from those which have been thrown away. The usual process is to search for all objects which could be referred to from the registers and the stack either directly or through other objects and call these the active objects. Anything else is certainly garbage because there is no way it can be addressed.

8.1 The Basic Scheme

The garbage collection process used in the Poly/ML system is essentially a compacting scheme but some variations. The basic scheme will be described first. The garbage collector works in three phases. The first phase finds the accessible objects and therefore the garbage. A simple-minded scheme would stop there, put the garbage on a free-list and the store allocator would allocate its store from that. The main disadvantage of this is that the store would become fragmented with lots of small free objects scattered through the store. Another problem is that searching a free-list is expensive. For these and other reasons the garbage-collector compacts the store so that the active objects form a contiguous area at the top of the storage region with the free area beneath it. The second phase compacts the store by moving active objects into store which was previously garbage, and the final phase changes the addresses of objects which have been moved so that they refer to their new locations.

The objects in the heap are represented by a bit-map with a bit for each word and initially the bits are cleared to zero. The first phase of garbage-collecting starts with the addresses in the registers and on the stack and sets the bits in the bit-map for each word in an object which it finds. This is called marking the objects and is a recursive process since each object may refer to other objects. An object is marked before the objects it refers to are checked and objects which have already been marked are not scanned again. This avoids problems with loops of references. Non-recursive marking algorithms have been proposed but have not been tried in the Poly/ML garbage-collector.

At the end of the first pass the bit-map shows the words which are known to be free and those which are garbage. The second pass starts from the bottom of the area and copies each active object up to the highest free area which will hold it. When it moves an object it leaves a "tombstone" behind which gives the address it has been moved to. At some point it will not be possible to copy an object any higher because there is no free store above it and the process stops with all the active objects compacted together at the top of the store. There may be a few small areas of free store left among the active objects if the last object was a large one, but measurements have shown that they are very few. The third phase

scans through all the addresses on the stack, in the registers or on the heap and changes every address which points to an object which has moved to point to the new location, using the "tombstone" which was left behind when it was moved. Finally the newly freed area is cleared to zero so that it can be reallocated.

The garbage-collector can only produce free store if there is some to be had and it is possible to have a program which keeps on creating objects and not throwing them away. The Poly/ML system starts with a relatively small amount of store for the heap and then gets more by calling the operating system if it runs out. It does this by counting the number of active words of store during the first phase of garbage-collection and then working out the amount which is free. If this has fallen below a certain amount it asks the operating system to extend the upper limit of the storage region to give some store to work with. This new area of store is treated as though it were free store so the second phase of the garbage-collection copies the active objects from the bottom of the heap up to this new area. The result is to make an area of store at the bottom of the heap available.

It might be thought better to have the active objects starting from the bottom the storage area and the free area at the top. New free store would then be added on at the end. The reason for not doing this has to do with the persistent storage system. For various reasons it is convenient for objects which have been brought in from the persistent store to be held in a separate area to the objects which have been created locally on the heap. The latter are often transient and so need to be garbage-collected while the former are longer-lived. Objects brought in from the persistent store are held in an area which extends from the bottom of the region upwards. This area grows up towards the heap as each new block brought in. When the areas meet the store must be garbage-collected.

8.2 The Full Scheme

The garbage collector as originally implemented worked as described above, but after some use it was found to have a serious drawback. Many programs, and in particular compilers, work by first building up a data structure such as a parse-tree and then processing it. While the data structure is being built up parts of it that have already been made are not actually being used. In a virtual memory system the part of the store containing them will be paged out. This works fine until the store has to be garbage collected. The scheme described must scan all the active objects at least twice, once during the marking, and again to find and update the addresses of objects that have moved. The effect is that every time the store is garbage collected all the objects in the virtual memory are paged in, and if real memory is tight this may involve a great deal of paging activity. The solution is to garbage collect only the part of the heap which is being actively used. The basic idea was used by Cardelli in his VAX ML compiler[4].

His scheme was primarily intended to reduce the cost of garbage-collection. He pointed out that objects varied in lifetime so that many newly created objects become free very quickly while others might survive several garbage-collections

before becoming free. In general the longer an object survived the lower the chance that it would be recovered during any particular garbage-collection.

The heap was divided up into regions by *age*. Initially objects were created in the youngest area, and as they survived more garbage-collections they migrated to the older areas. The younger regions were garbage-collected frequently and the older areas less frequently.

When a region is garbage-collected all the addresses into it must be identified in order to mark the objects and then they must later be updated after the objects have been compacted. If this required searching the whole of the heap to find the addresses there would be no advantage in garbage-collecting only a single region, but it is possible to restrict the search to only part of the heap.

Ignoring for the moment updatable values, (*references* in ML or *variables* and *arrays* in Poly), this is not too difficult. Whenever a new object is created it can only refer to objects that already exist. This means that an object in a particular region can only contain references to objects in the same region or an older one. A particular region can be garbage-collected by looking for references to it from the younger regions only, the older regions need not be touched. Using this method the garbage collector does not need to look at the older objects and the virtual memory activity is much reduced.

So far we have ignored the problem of updatable values. If a variable can be assigned to after it has been created then it is possible for it to refer to an object newer than itself, and if the variable is in an older region and the object it refers to is in a newer then the variable would have to be changed if the compaction phase moved the object. To avoid this problem updatable objects are put into a separate area of memory , the *local mutable area*, which is scanned during the marking and updating phases to find addresses every time any region is garbage collected. Because updatable values are relatively uncommon in Poly and ML this extra cost is quite small. The *local mutable area* can only itself be garbage-collected when the whole of the store is garbage-collected.

The actual implementation divides the heap up dynamically rather than having static regions. The heap grows downwards from the top of store with the topmost area used as the *local mutable area*. Below this is the oldest region with newer regions lower down the heap. The lowest area, immediately above the persistent store region, is used for newly created objects. New objects, whether updatable or not, are created in this region, but updatable objects are marked with the *mutable* bit set. When garbage-collection is necessary the newly created objects, and possibly some of the other regions, are together garbage-collected using the scheme described before. The result is a single region containing the youngest objects. During the compaction phase any updatable objects are moved into the *local mutable area*. Periodically the whole of the store is garbage-collected, particularly when more store is obtained from the operating system, since this is allocated at the top of the store.

Figure 8 shows the local area of the heap immediately before a garbage-

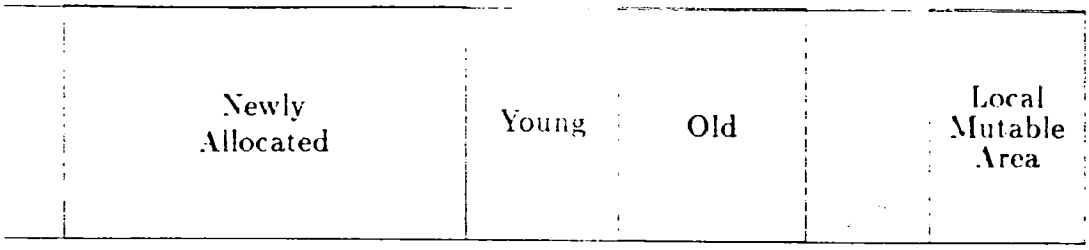

Figure 8: Before Collection

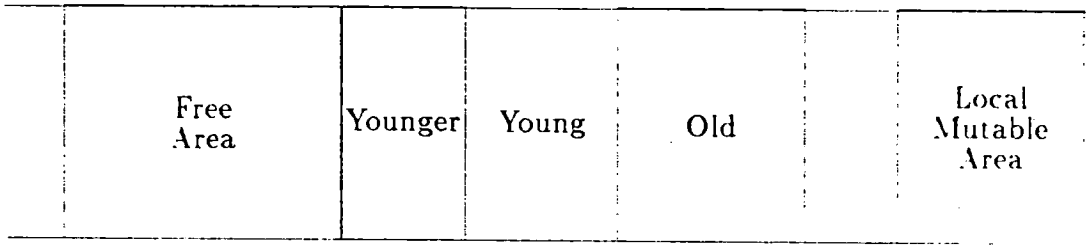

Figure 9: After Partial Garbage Collection (1)

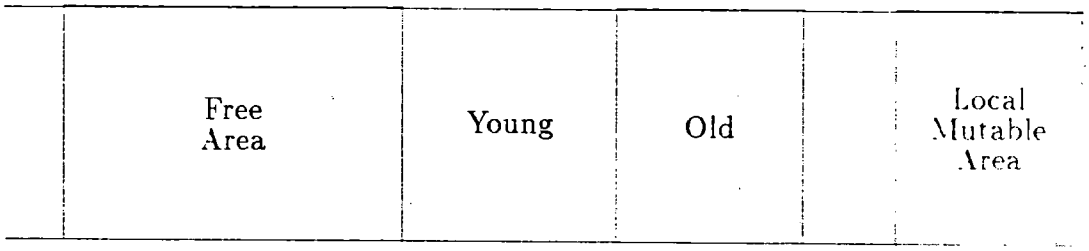

Figure 10: After Partial Garbage Collection (2)

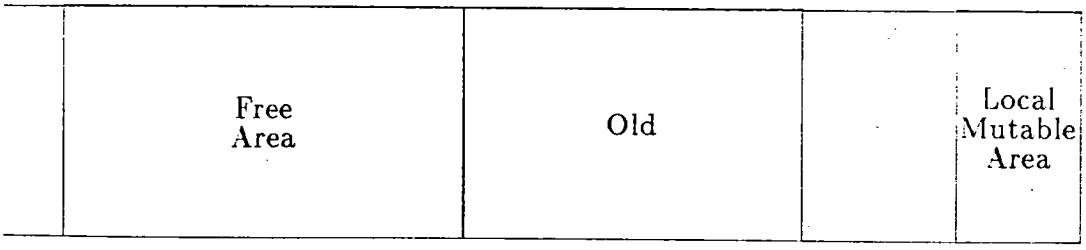

Figure 11: After Full Garbage Collection

collection. Two regions, "Old" and "Young" are shown. Figure 9 shows the area after the newly allocated objects have been collected into a new region, "Younger". The mutable area has increased in size slightly as newly allocated mutable objects have been copied into it. Figure 10 shows an alternative partial collection in which "Young" has been included in the collection. Again the newly allocated mutable objects have been added to the mutable area. Finally Figure 10 shows the state after a complete garbage-collection. The immutable objects have been compacted into a single "Old" area, and the mutable area, which has not been garbage-collected since the last full collection, has now been reduced in size as short-lived mutable objects have been collected.

As well as *references* or *variables* it is possible for an object to refer to objects newer than itself. This can occur in the case of, for example, the closures of mutually recursive procedures, where each closure may refer to the other. These will only cause problems if the store is garbage-collected after one closure has been created but before it has been filled in with the reference to the other. To avoid these problems a newly created object is marked as updatable and then this mark is removed when the object has been filled in.

In general all the words in a newly allocated object are filled in with values immediately after it is created. However in some cases, for example with closures of mutually recursive procedures, the garbage-collector could be called before the whole of an object has been filled in. For this reason after each garbage-collection the free store is cleared so that unset words are always zero. I found that clearing the whole of the free area immediately after a garbage-collection caused a great deal of virtual memory activity as pages were read in to be cleared and then written out again. One solution would have been to clear each object when it was allocated but this would make the allocation process more complicated, and since the code for allocating an object is compiled in-line it would increase the size of the code. Instead a compromise was found whereby only part of the free area is cleared and the garbage-collector is called when that area is used up. If there is more space available the garbage-collector simply clears some more space and returns, and only actually garbage-collects if the space is used up. This scheme reduces the paging activity since the pages that are cleared are likely to be used for new objects fairly quickly.

9 External Interface

The Poly code must be able to call functions which provide an interface to the outside world. It would be possible to insert code to do system calls directly in the Poly code using some sort of escape mechanism but it is better to isolate the operating system dependencies in a separate run-time system, leaving the Poly code independent of the operating system.

The run-time system is largely written in C, making operating system calls fairly simple. Rather than have the Poly code call these directly it indirects

through an interface map. In this way the addresses of run-time system routines are not bound directly into the Poly code so changes can be made to the run-time system without requiring the Poly system to be reconstructed. The entries in the vector have special persistent addresses and the persistent storage system maps these onto vector entries when objects are read in.

Another problem is that the calling conventions of C and Poly are different so there is a piece of assembly code for each function which converts the arguments and results between the formats.

References

[1] Robin Milner, *The Standard ML Core Language*, Edinburgh University.

[2] David C.J. Matthews, *The Poly Manual*, SIGPLAN Notices, September 1985.

[3] David C.J. Matthews, *A Persistent Storage System for Poly and ML*, Technical report 102, University of Cambridge Computer Laboratory, January 1987.

[4] Luca Cardelli, *The Functional Abstract Machine*, Computing Science Technical Report No. 107 AT&T Bell Laboratories.

Interfacing C Routines to Poly/ML

David C.J. Matthews

16 December 1987

1 Abstract

ML is a powerful programming language and you can do almost anything in it you could do in any other language. The exceptions are where you want to use some feature of the operating system that has not been provided in the standard libraries, perhaps because it is specific to one operating system. In that case you might want to add a C routine to the run time system to call that function. Alternatively you may have a piece of code already written in another language which you want to be able to use from ML.

This report describes how to interface C routines to ML, and the problems you may encounter.

2 The Simple Method

Suppose you have a routine written in C, or perhaps some other language, which you want to be able to call from ML. The simplest way of doing this is to make it into a complete program, and then run it using `ExtendedIO.execute`. This starts the program and returns streams connected to its standard input and standard output. You can send arguments to it down one stream and get results from the other one. You can provide it with a suitable functional interface so that it appears to the ML as an ordinary ML function.

3 Adding a Function to the Run-Time System

That is by far the easiest way of interfacing a C routine because it avoids any need to know about the internals of the Poly/ML run-time system. However, if you really want to incorporate your C routine directly into the driver program for Poly/ML this is how to do it.

The first step is to add your C routine into the driver. You can edit it into `run_time.c` which is just a collection of assorted run time functions, or put it as a separate module and link it in. In the latter case you will have to change `Makefile` to compile and link it. Each function you want to call from ML has to

have a name which ends in the letter "c". So we will assume that the function you want to call is `my_fun_c`.

The next stage is to find a *function number* for your function. This will enable the ML code to find your C function. You do this by looking at the entries in `globals.h` which begin `SYS_`. In the standard system the numbers up to 255 are used with some gaps for future enhancements. To avoid problems with future releases use 256 or above. You must increase `SYS_vec_size` to be at least one more than the number you have used. You should add an entry for your function.

```
#define SYS_my_fun 260

#define SYS_vec_size 261
```

You now need to edit the appropriate machine dependent files. The Vax and Sun versions, and others as they are produced, each have a pair of files in the run-time system that deal with all the machine dependent parts. The rest of the run-time system is common to all versions. The two machine dependent files are a C source and an assembly code source. If you are running on the Sun, for example, the files are `sun_dep.c` and `sun_assembly.s`, and on the Vax they are `vax_dep.c` and `vax_assembly.s`. You will have to make changes in both of these files.

In the C source you must add an entry to the `interface vector`. You do this by editting the routine `MD_init_interface_vector` and the declarations immediately above it. You must add a declaration of your C function but replacing the final "c" by an "a".

```
extern word my_fun_a;
```

Inside the function you add a line which looks like

```
add_word_to_io_area(SYS_my_fun, &my_fun_a);
```

You can put it anywhere among the other calls, they are in no particular order.

Next you must edit the assembly code file. There are macros to help you here. You will find a set of lines which look like

```
CALL_IO1(convlong, REF, IND)
CALL_IO0(processtime, IND)
CALL_IO2(openstream, REF, VAL, IND)
```

You must add a similar line for your function. Each line is a macro which expands into a piece of assembly code which will take the parameters supplied by the ML call and pass them to the C function, and then take the result of the C function and pass it back to the ML.

The number immediately after `CALL_IO` is the number of parameters, 0 if the function has no parameters. The first of the arguments to the macro is the name of your C function, but without the final "c". After this come entries which describe the way that the parameters should be passed to your C function, either as the

values themselves (VAL), or by reference (REF). There must be one of these for each of the parameters to your function, and they are given in the order in which the parameters are passed from ML which is the reverse order to the order of the parameters of your C procedure. Finally there is an argument to the macro which describes whether the result of the C function should be returned directly to the ML (NOIND) or whether the result of the C function is the address of the actual result (IND). So if your function has two parameters, the first of which is passed by reference and the second by value, the result being returned directly, you need an entry which looks like

```
CALL_IO2(my_fun_, VAL, REF, NOIND)
```

When do you use VAL, REF, IND and NOIND? VAL and NOIND can be used when the argument or result is a boolean or an integer that you know will be between -32768 and 32767. If it is a structured value, a string or an integer outside this range you will almost certainly have to use REF or IND. If you use REF your C routine will be given the address of the argument so you may have to change your C routine to accept this. Similarly if the result is IND you may have to change the way you return the result.

Now you have made all the changes to the run-time system you can remake the driver program. Assuming that works successfully you can use this new driver with your existing database and put in the ML function which will call your C routine. You need to put in an ML function with the appropriate type. You use a function from the PolyML structure which provides the interface and make a declaration of the form

```
val my_fun : string * int -> bool = PolyML.run_call2 260
```

The number 260 was the number used for SYS_my_fun in globals.h, and run_call2 is used because the function has two arguments. A type coercion must be given to give your function the correct type. The type you give it is up to you but remember that the arguments are in the reverse order to the C routine.

You cannot actually type in the ML as shown because the PolyML structure that is normally available does not have the run_call functions. They provide a loophole in the type system and the run-time system so they are deliberately made difficult to find. What you must do is put the declaration in a file, say my_fun.ML. You now have to enter the poly system and compile the file by using the following sequence of commands.

```
PolyML.poly();
runcompiler(ml_environ, ml_compiler, new(root_environ), ".ML")
    "my_fun";
^D
```

You can only do this in the top-level (parent) database and not in a database constructed with make_database. You will now be back in ML and you can try out your function.

4 Complications

That was the easy part. Unfortunately it is probably not going to be as easy as that. If your C routine is very simple, for example just calling an operating system function, you may be able get away with using it as it stands, otherwise you are probably going to have to make some changes to it to make it work with the rest of the run-time system.

The problem is that addresses of ML objects may be persistent addresses, and therefore cause a bus error if you use them as C pointers, or the objects they reference may move as a result of a garbage collection. This is the reason why `REF` is used to pass objects that may be addresses. A value passed as `REF` is put into the `save vector` and the value passed to the C routine is the address of the entry. The garbage collector knows about values in the save vector and will ensure that the objects they refer to are not thrown away, and that the addresses are updated if the objects move. Provided you always indirect through the vector you will be safe.

If the values passed in to your C routine can be addresses they may well be in the persistent form. To ensure that they are in the local form and to load the object if necessary there is a C routine called `load_persistent_object` which takes an address as an argument and returns it in the local form.

If you want to create an object on the heap you can call `alloc_and_save`. This takes as argument the number of words to allocate and allocates a piece of store, possibly after a garbage collection. The address of the store is put into the save vector and the address of the entry is returned. In this way multiple calls can be made without the results of previous calls being garbage collected. If you want to return such a value to the ML code you can return the address of the save vector entry and use `IND` in the `CALL_IO` macro.

Because strings are often passed into and out of the run time system there are two routines, `Poly_string_to_C` and `C_string_to_Poly` which take care of the conversion of formats, and any persistent address translation.

Putting addresses in the save vector is satisfactory if you only want them during a single call of your function. But sometimes you may want to be able to keep some information from one call to the next in a static variable. You can do this, but you must be careful if the values include addresses of objects on the heap. You will have to register a procedure with the garbage collector so that it will not throw your objects away. You do this by calling `register_gc_proc` with your procedure as the argument.

```
register_gc_proc(gc_proc)
gc_proc(update_op)
update_op(ptr, strength)
word **ptr;
int  strength;
```

The garbage collector will call `gc_proc` (your procedure) with a procedure to apply

to the address of each of your addresses, `update_op`. The procedure you are given will update the address if the garbage collector moves the object. As well as the address it takes a `strength` value which is either 0, for a *strong* reference, or 1 for a *weak* reference. Strong references are the normal case and weak references are only used in exceptional circumstances. If you pass an address with the strength set to 1 the garbage collector may set the address to zero if there are no other references to the object. This can be used, for example, to close streams which are no longer referenced.

C programs often build up structures using calls to `malloc` and get rid of them with calls to `free`. The Poly/ML run time system has redefined these so that the store allocation will work with the rest of the system. The objects allocated by `malloc` are not moved by the garbage collector, but calling `malloc` may cause a garbage collection in order to find the space. Addresses allocated by `malloc` calls must not be passed back to the ML system directly since they do not follow the conventions of ML addresses. Instead you will have to put them into a static vector and return an index into it. In that way the ML program is given a small integer as the result of the call and this can be used as an argument to other calls. If you do this kind of thing you will have to think about what will happen if the user saves their database and then restarts in another session. Clearly the structures created with `malloc` will not be there in a new session.