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 ˆD). 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.

$$poly\quad -r\ \ poly\_dbase$$

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 (ˆD). 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 recompiling 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