

Using ML as a Command Language

Steve J. Chapin Ryan Stansifer

Department of Computer Sciences

Purdue University

West Lafayette, IN 47907

sjc@cs.purdue.edu ryan@cs.purdue.edu

1 Introduction

We consider using a typed, functional language as the command line interpreter. Often the user interface to the operating system, or shell, is ad hoc in design. There is an increasing tendency to add on baroque extensions to shells, resulting in an inconsistent and confusing command language. Frequently these extensions are constructs that have analogs in programming languages. In the interest of regularity it seems appropriate to move towards a monolingual environment, i.e., one in which the user issues commands and constructs programs in the same language. In order to implement such a shell, we choose an existing programming language, ML. ML is a logical choice due to its conceptual simplicity and perspicuity.

In this paper, we examine the purpose and use of shells as the interface to UNIX¹. We determine which features of shells are already present in ML, and which need to be added. We are modifying the underlying abstract machine for our own ML interpreter to include these capabilities as an integral part of the language.

2 What Is a Shell?

The user interface to the operating system is usually in the form of a command line interpreter. The following list enumerates the capabilities generally associated with command line interpreters. Most shells will:

1. the ability to invoke arbitrary programs named by the user as processes.
2. provide control structures, such as looping and conditionals, for the combination of shell primitives.
3. interpret programs written in the command language, also known as *shell scripts*.

¹UNIX is a trademark of AT&T.

4. run tasks asynchronously, allowing tasks to run concurrently. with the shell, as well as providing some control over the execution of these tasks.
5. compose programs to form more complex functions via the connection of the output of one program to the input of the next.
6. perform I/O redirection, allowing files to be used in place of input and output devices.
7. allow the user to customize the operational environment for programs the shell will invoke.
8. provide reasonable performance and a concise syntax.
9. perform path searching, wildcarding, and filename completion. Path searching is the act of looking for a named file in a list of directories, often in anticipation of executing the file. Wildcard expansion refers to the ability of the shell to translate an expression representing several file names into the actual names. Filename completion allows the user to type a unique prefix of the name and have the shell fill in the rest.
10. provide a history mechanism to allow the user to repeat often used commands easily.

The UNIX shell delivers a remarkable amount of power for a minimum effort. Its notion of a stream of characters as the only I/O data type makes function composition intuitive and relieves the programmer of type considerations.

A major advantage of using an existing programming language as the basis for a command interpreter is that we gain all the flexibility and power of the existing language. As noted in [HK85], the user has only one language to learn for both programming and for issuing commands. Most programming languages, including ML, have features that are analogous to items 2, 3, 4, and 5. The other items will have to be added to the ML interpreter if it is to act as a shell. We have available the ML conditional and iteration structures, structured data types, and the ability to write scripts to perform complex tasks.

We therefore will combine the preferable aspects of the UNIX shell with the ML programming language. The result will have the versatility of the shell with ML's powerful typing system and control structures.

3 Operating System Service Requirements

Before defining the shell primitives that will allow a user to access operating system services, we must first determine which services are of interest. This information will help guide our selection of primitives and their semantics.

Some of the items listed in section 2 are operating system issues. Both the ability to run an arbitrary program and the composition of programs fall into this

category. Input/output redirection and job control also require operating system access.

Several common shell features require knowledge of and access to the file system. A function similar to the UNIX `stat` system call that reports information on a selected file will be of use for writing the ML function to do filename completion, wildcarding, and path searching.

The concept of the *working directory* as a reference point in the file system for relative path operations, such as a call to `open_in "datafile";` is important. Since a complete path name is not specified, the file `datafile` is opened relative to the working directory. The shell needs a mechanism to set this working directory for itself. Although UNIX provides a default mechanism to propagate the working directory to child processes, it would be useful to allow the shell to explicitly set the working directory for a child process.

When a file is created, it usually inherits a default set of access rights. In UNIX, this is called the `umask`, and the shell should be able to set the value of this operating system variable.

4 The CML Shell

CML is an implementation of the Core ML language, written at Purdue University. We are extending CML to make it suitable for use as a command interpreter. One very important feature is the stream data type, as presented in [Har85]. This solves the problem that ML has with emulating a command interpreter on the computation of function values in composition. A stream of characters can be thought of as a possibly infinite sequence of characters. Streams come in two types: `instream` are streams that can be read, and `outstream` can be written. Our ML shell has two predefined data types for streams, `instream` and `outstream`.

Without the stream type, ML must wait for the value of a function to be completely computed before it can pass that value to another function; e.g. in the function composition $f(g())$, ML must wait for g to complete computation before it knows the value of the argument for f . Streams allow ML to pass partially computed sequences of characters between functions.

We have also defined several functions for file I/O, as listed in table 1.

Several of the upcoming examples will use the definition

```
val sio = (std_in, std_out, std_err);
```

It is a notational convenience that represents the common UNIX triple of the standard input stream, the standard output stream, and the standard error stream. We intend to provide access to operating system features via extended CML primitives and to allow the programmer to use the ML language features to customize the interface to these features. The following sections contain examples of these extensions.

item	description	type
<code>open_in</code>	function to open an input file	<code>string -> instream</code>
<code>open_out</code>	function to open an output file	<code>string -> ostream</code>
<code>open_append</code>	function to open an output file in append mode	<code>string -> ostream</code>
<code>close_out</code>	function to close an ostream	<code>ostream -> unit</code>
<code>close_in</code>	function to close an instream	<code>instream -> unit</code>
<code>std_in</code>	standard input	<code>instream</code>
<code>std_out</code>	standard output	<code>ostream</code>
<code>std_err</code>	standard error	<code>ostream</code>

Table 1: Input/Output features of CML

4.1 exec

CML has an `exec` function of type

```
string ->
  instream * ostream * ostream ->
    (string * string) list ->
      string ->
        unit
```

The first argument is a string representation of a UNIX program name to be run, the second is a triple of streams for input and output, the third is the environment list, and the last is a string containing the arguments for the program. An example might be

```
exec "/bin/ls" sio environ "foo.c";
```

`exec` would locate the file `/bin/ls` in the file system, execute it and return `unit` if it could find the file, otherwise it would raise an exception. The `environ` argument is a list of name/value pairs comprising the UNIX program's environment.

If we were to make the definitions

```
val ls = exec "/bin/ls" sio environ;
fun ll (args) = ls ("-l " ^ args);
```

we could then pass arguments with short statements such as `ls "foo.c"`; or `ll "foo.c"`, giving an easy shorthand while still allowing the user the ability to construct complicated commands if required.

The definitions of our shell primitives are intended to allow the programmer the maximum control over the shell functions, while using ML code to build higher level functions. This leaves the programmer with a wide range of choices for power and convenience.

```

val path = [ ".", "/usr/sjc/bin", "/bin", "/usr/bin" ];

fun docmd cmd io env s =
  let
    val cmd2 = resolve_cmd(cmd);
    val args = expand_args(s);
  in
    exec cmd2 io env args
  end;

fun ls = docmd ls sio [];

ls "*c";

```

Figure 1: Example use of the `exec` function

The `exec` function is a good example of this philosophy. `exec` does not itself do any path searching or wildcard expansion with its arguments. We have left these mechanisms to be defined at a higher level (`resolve_cmd` and `expand_args` above), so that the user can customize the shell. Default functions are provided to perform these tasks. An example of using the power of the `exec` function while retaining flexibility is in figure 1.

4.2 Error Handling

UNIX error handling is primitive in comparison to ML error handling. By convention, UNIX processes return an integer code, with 0 representing successful completion, and any non-zero value indicating an error. `exec` could be written to return an integer instead of the unit value, but to encourage the use of the exception mechanism provided in ML, `exec` raises an exception when the UNIX process returns a non-zero value.

4.3 pipe

In order to aid function composition between UNIX programs, we have added the `pipe` function to CML. It is of type `unit -> instream * outstream`. Given the standard form for UNIX commands, we can then define a piping mechanism shown in figure 2. Note that `pipe` returns a pair of streams that are connected, and that `Pipe` uses this call to connect successive commands together. We place the I/O streams as the first argument to both the UNIX commands and the `Pipe` call. This allows us to curry these operators to achieve a desirable conciseness of syntax.

```

fun Pipe (s1, s2, s3) env (f, a1) (g, a2) =
  let
    val (i,o) = pipe();
    val () = docmd f (s1, o, s3) env a1;
    val () = docmd g (i, s2, s3) env a2;
    val () = close_in(i);
    val () = close_out(o);
  in
    ()
  end;

Pipe sio environ;

```

Figure 2: sample ML definition and use of piping function

4.4 Shell Variables

In the past, there has been some confusion in the treatment of shell variables in command interpreters. We feel that there are two kinds of variables that a shell must deal with, intended for completely different purposes. First, there are variables local to the shell that are not intended to affect the shell's children; second, there are variables that are used to modify the environment of child processes but which are not needed by the shell. The second type of variable is often referred to as an environment variable.

Past shells have used a lookup mechanism that has clouded this distinction. Most shells will look for a variable in the environment if it is not found locally. Our shell keeps these two issues separate by using ML variables for local variables and making the environment for children an explicit parameter of the `exec` call.

The environment is intended to model the UNIX environment, supplying a list of variables that a program can access to customize the interface for the user. The shell's environment can be accessed through a predefined constant `environ`, which has type `(string * string) list`. Since we explicitly name the environment for all UNIX commands, it is easy to supply a different environment to commands by passing a modified copy of the local environment. Therefore, we can simulate a subshell by using a `let` statement to modify the environment that will be passed to UNIX commands; this will not affect our local environment.

5 Conclusion

We have shown that ML can be used as the basis for a command interpreter for UNIX. A functional shell has many advantages over a conventional shell, and ML provides much of the mechanism for implementing such a shell at essentially no cost. We feel that the gain in expressive power far outweighs the slight degradation in speed, and note that a distributed implementation of ML could provide an increase

in speed substantial enough to offset the additional overhead our shell incurs.

Although it is not the primary goal, we intend to add many of the convenience features of other command line interpreters to CML. The `readline` package distributed by the Free Software Foundation implements emacs-style command editing and history mechanism for general use.

CML has all of the essential features of a command line interpreter except job control. The addition of job control would require a new data structure for the shell containing pertinent information about jobs such as their status, name, and arguments. This could be bound to the symbol `jobs` in the initial environment. Job suspension is a desirable feature, and could be invoked using the UNIX convention of control-Z to cause an interrupt and suspend the currently executing process, returning control to the shell. We would also need commands to change job state (e.g., `suspend`, `foreground`, `background`). The shell would have to be modified internally to handle the SIGTSTP signal and to do the correct I/O handling.

This paper is a condensed version of [CS90]. [AM87] describes the Standard ML of New Jersey compiler which has some shell features. An implementation of the UNIX shell written in Standard ML appears in [KM87].

References

- [AM87] Andrew W. Appel and David B. MacQueen. A Standard ML Compiler. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer-Verlag, 1987.
- [Bou] S. R. Bourne. *An Introduction to the UNIX Shell*.
- [CS90] Steve J. Chapin and Ryan Stansifer. Using ML as a Command Language. Technical report, Purdue University, January 1990.
- [Har85] Robert W. Harper. Standard ML Input/Output. Technical report, Edinburgh University, June 1985.
- [HK85] J. Heering and P. Kint. Towards Monolingual Programming Environments. *ACM Transactions on Programming Languages and Systems*, April 1985.
- [Joy] William Joy. *An Introduction to the C shell*.
- [KM87] Yogeesh H. Kamath and Manton M. Matthews. Implementation of an FP-Shell. *IEEE Transactions on Software Engineering*, SE-13(5):532–539, May 1987.
- [Kor] David Korn. *Introduction to KSH (Issue 2)*.
- [McD89] Cristopher S. McDonald. An Executable Formal Specification of a UNIX Command Interpreter. In *IFIP W2.7 Working Conference on Engineering for Human-Computer Communication*, August 1989.