Denotational Semantics of a Command Interpreter and their Implementation in Standard ML

C. McDONALD[1]* AND L. ALLISON[2]

[1] *Department of Computer Science, The University of Western Australia, Nedlands, Western Australia, 6009*

[2] *Department of Computer Science, Monash University, Clayton, Victoria, Australia, 3168*

Several working groups have been established to study and standardize popular command interpreters. However, as with the standardization of many programming languages, the syntaxes of command interpreters have been rigorously defined in notation such as Backus-Naur Form (BNF) but the semantic definitions remain ambiguously defined in natural languages such as English. This paper defines a significant subset of the standard UNIX command interpreter, or shell, in terms of its denotational semantics. A complete implementation of this shell in Standard ML is described. This implementation enables direct execution of the denotational semantics and encourages experimentation with the semantic definition.

Received July 1988

1. INTRODUCTION

A number of working groups have been established to study and define command interpreters and their interactions with operating systems. These groups have reported on the standardization of command interpreters available on existing operating systems,[1] attempts to define command languages that are portable across different operating systems[2] and reference models which specify the interactions between components of command and response languages.[3] Such groups have been successful in producing natural language definitions of command interpreters. Implementation of command interpreters from these definitions alone is seen as a goal of the working groups.

Command interpreters are seen as central tools linking together independent programs. They perform two important functions in an interactive environment. Firstly, they permit a user to control their environment by defining the type and availability of commands to execute. Secondly, they permit the execution and control of commands based upon the user's environment and the results of previous commands. In efforts towards providing monolingual programming environments, facilities more traditionally associated with programming languages have been added to command interpreters.[4] It is now seen as a requirement of interactive command interpreters that they support features such as conditional execution, iteration and the definition of user-defined commands.

Whereas there has been significant effort in defining both the syntax and semantics of programming languages using formal notations, there has been little corresponding work in defining command interpreters.[5] Madsen has described the Catalogue management routines of the job control language for IBM's OS/360.[6] Despite the relatively small responsibility of the Catalogue routines, Madsen successfully demonstrates that operating system facilities can be defined formally. Formal specification of another aspect of operating system facilities has been provided by Morgan in defining the UNIX filing system using set notation.[7] Currently the IEEE P1003.2 POSIX Working Group is defining the 'Shell and Tools Interface' of a portable operating system. The work of this group is expected to take two years and produce a definition of a command interpreter based on the UNIX Bourne shell. Although this definition will include a significant specification of the interpreter's syntax and semantics, no plans for the use of a formal notation such as denotational semantics are known to exist.

In this paper we define a significant subset of a UNIX command interpreter, or shell, in terms of its denotational semantics. Our interpreter supports the traditional environment structure, input and output (I/O) redirection, process creation and conditional and iterative control flow mechanisms of the standard UNIX command interpreters. However, issues such as textual aliasing, command revision and job control are not addressed. We have also made no effort to define the concrete syntax of our command interpreter, choosing only to define the abstract syntax. Our command interpreter is based on the popular Bourne and C shells. There are concrete syntax differences in these shells, such as the syntax for setting environment variables, I/O redirection and conditional and iterative control flow. We ignore these differences here, believing these structures to be reducible to a common abstract syntax. Subsequently, we have defined the denotational semantics of significant subsets of both popular shells.

The notation used for our semantic definition is based on the notation defined by Stoy.[8] Having defined the denotational semantics of our interpreter we have implemented the definition in a metalanguage. Rather than using either Pascal or Algol as the metalanguage, as demonstrated by Allison,[9] we have chosen the functional language Standard ML (SML).[10] The significant advantages found with this approach will be described. Our implementation both provides verification of the syntactic specification of our semantics and enables direct execution of these semantics.

2. DEFINITIONS

The definition of a programming language in terms of its direct denotational semantics involves four distinct steps. These are the definition of the language's syntactic

domains, syntactic clauses, semantic domains and semantic equations.[11] With few deviations, these steps may be followed in defining a denotational semantics of a UNIX shell.

2.1. Definition of Syntactic Domains

The abstract syntax of a programming language usually defines three *syntactic categories*: declarations, commands and expressions. For each of these syntactic categories a *syntactic domain* is defined for all possible constructs of that category. We define our UNIX shell with three syntactic domains named **Args, Redirection** and **Command**. The *metavariables* α, ρ and γ and subscripted instances of these are used to range over each domain respectively.

2.2. Definition of Syntactic Clauses

Syntactic clauses form an *abstract* syntax defining all valid components for each syntactic domain. Note that the abstract syntax does not unambiguously define a syntax on which a practical parser could be constructed. For example, the relative binding strengths of the infix operators are unspecified in the abstract syntax. Each `if` clause requires an `else` clause and each *simple command* requires input/output redirection. Disambiguating rules required to construct a practical parser need to be defined with a *concrete* syntax.

$\rho ::= \phi_\rho$	*empty redirection*
$\mid < \alpha\rho$	*input redirection*
$\mid > \alpha\rho$	*output redirection*
$\mid \; \alpha\rho$	*append output*
$\mid > \& \; \alpha\rho$	*error output*
$\mid \; \& \; \alpha\rho$	*append error output*
$\gamma ::= \phi_\gamma$	*empty command*
$\mid \alpha\rho$	*simple*
$\mid (\gamma)\; \rho$	*subshell*
$\mid$ `if` γ_1 `then` γ_2 `else` γ_3	*conditional*
$\mid$ `while` γ_1 `do` γ_2	*iterative*
$\mid$ `until` γ_1 `do` γ_2	*iterative*
$\mid \gamma_1 ; \gamma_2$	*sequential*
$\mid \gamma_1$ && γ_2	*'and' conditional*
$\mid \gamma_1 \parallel \gamma_2$	*'or' conditional*
$\mid \gamma_1 \& \gamma_2$	*asynchronous*
$\mid \gamma_1 \mid \gamma_2$	*pipe*

Equation 1. The Syntactic Clauses

2.3. Definition of Semantic Domains

We define the denotational semantics of our shell with two semantic domains. The **State** domain represents the interface between the shell and the operating system. Each process under the UNIX operating system has associated with it a number of attributes which describe and constrain the process. These typically include resource allocations and limits, the relationship of the process to other processes and the current execution status of the process. In defining the **State** domain for the shell, it is only necessary to define attributes over which the shell need have control.

Our **State** domain defines four input and output streams through which the shell communicates with the operating system. Each I/O stream is represented by a possibly infinite sequence of bytes. The shell also supports user-defined variables which are maintained in an association list of string values referred to as the *environment*. The use of the word 'environment' should not be confused with the concept of the declaration environment in a traditional programming language. Elements of the shell's environment are tagged as either exported, which are made available to subprocesses, not exported, which are available only within the shell and read only, which reflect the shell's internal operation and are not exported. The shell's **State** is analogous to the *global store* of a program during execution.

On termination, each UNIX process returns an indication of its success to the operating system. This *exit status* is also available to the process that invoked the terminating process. The exit status is typically represented as an integer with the value zero indicating success and any other value indicating some error condition. We use the **Exit** domain to represent this exit status.

The *metavariables* ε and σ and subscripted instances of these are used to range over the **Exit** and **State** semantic domains respectively. The *semantic domains* of the shell are defined in Equation 2.

$$
\begin{aligned}
\varepsilon\text{: }&\textbf{Exit} = \texttt{Integer}\\
\sigma\text{: }&\textbf{State} = \texttt{I/O} \times \texttt{Env}\\
&\texttt{I/O} = \texttt{Input1} \times \texttt{Input2} \times \texttt{Output1} \times \texttt{Output2}\\
&\texttt{Env} = (\texttt{Name} \times \texttt{Value} \times \texttt{Envtype})^*\\
&\texttt{Input1} = \texttt{Byte}^*\\
&\texttt{Input2} = \texttt{Byte}^*\\
&\texttt{Output1} = \texttt{Byte}^*\\
&\texttt{Output2} = \texttt{Byte}^*\\
&\texttt{Envtype} = \texttt{Export} + \texttt{NoExport} + \texttt{Readonly}
\end{aligned}
$$

Equation 2. The Semantic Domains

2.4. Definition of Semantic Equations

Semantic equations define the denotations of constructs in the abstract syntax. For each construct we define corresponding mappings from initial instances of the semantic domains to possibly modified instances. We begin by defining a *valuation function* for each syntactic construct. Each valuation function is further defined by case analysis. For the **Args, Redirection** and **Command** domains we define:

$$
\begin{aligned}
\mathrm{EA} &= \lambda(\alpha \times \sigma_1) \rightarrow (\varepsilon \times \sigma_2)\\
\mathrm{ER} &= \lambda(\rho \times \sigma_1) \rightarrow (\varepsilon \times \sigma_2)\\
\mathrm{EC} &= \lambda(\gamma \times \sigma_1) \rightarrow (\varepsilon \times \sigma_2)
\end{aligned}
$$

Equation 3. The valuation functions for the Syntactic Domains

The definition of each construct in the **Command** domain is grouped with similar constructs. Unlike conventional programming languages, the UNIX command interpreters do not support a **Boolean** domain for use in conditional evaluation. Instead, instances of the **Exit** domain are used to determine required evaluation. An **Exit** value of zero is analogous to *true*. The syntactic constructs ; , && and || are interpreted as 'and then', 'if then' and 'if not then' respectively. The & construct defines asynchronous evaluation of two commands. The shell invokes a separate instance of itself to evaluate the first command. The original instance of the shell evaluates the second command and uses this result as that of the equation. The 'sequential execution' group of EC equations are:

```
EC⟦γ1 ; γ2⟧ σ1 =
  let ⟨ε1, σ2⟩ = EC⟦γ1⟧ σ1
  in EC⟦γ2⟧ σ2
EC⟦γ1 && γ2⟧ σ1 =
  let ⟨ε1, σ2⟩ = EC⟦γ1⟧ σ1
  in (if success ε1 then EC⟦γ2⟧ σ2 else ⟨ε1, σ2⟩)
EC⟦γ1 || γ2⟧ σ1 =
  let ⟨ε1, σ2⟩ = EC⟦γ1⟧ σ1
  in (if success ε1 then ⟨ε1, σ2⟩ else EC⟦γ2⟧ σ2)
EC⟦γ1 & γ2⟧ σ1 =
  let ⟨ε1, σ2⟩ = SETINPUT2⟦'/dev/null'⟧ σ1
  in (if success ε1 then PARALLEL⟦γ1 γ2⟧ σ2 σ1 else ⟨ε1, σ2⟩)
```

Equation 4. The sequential execution constructs of the EC function

The valuation function for the | construct requires the creation of a temporary I/O stream, or *pipe*. The pipe connects the output stream of the left hand **Command** to the input stream of the right hand **Command**. The two **Commands** are then evaluated asynchronously. The exit status of the | construct is defined to be the exit status of the right hand **Command**. A more detailed examination of the formal semantics of UNIX pipes is given in Ref. 12.

```
EC⟦γ1 | γ2⟧ σ1 =
    let ⟨ε1, σ2⟩ = PARALLEL⟦γ1 γ2⟧ σ3 σ4
      where π = PIPE
      and (... × in2 × π × ...) = σ3
      and (... × π × out1 × ...) = σ4
    in ⟨ε1, σ1⟩
  where (... × in2 × out1 × ...) = σ1
```

Equation 5. The | construct of the EC valuation function

The EC function also defines conditional and iterative sequences in the shell. As with the conditional operators, the exit status of a command is compared against zero to determine which branch of a condition is to be executed or if iteration is to be continued.

```
EC⟦if γ1 then γ2 else γ3⟧ σ1 =
    let ⟨ε1, σ2⟩ = EC⟦γ1⟧ σ1
    in (if success ε1 then EC⟦γ2⟧ σ2 else EC⟦γ3⟧ σ2)
  EC⟦while γ1 do γ2⟧ σ1 =
    let ⟨ε1, σ2⟩ = EC⟦γ1⟧ σ1
    in (if success ε1 then (let ⟨ε2, σ3⟩ = EC⟦γ2⟧ σ2
                            in EC⟦while γ1 do γ2⟧ σ3)
        else ⟨ε1, σ2⟩)
EC⟦until γ1 do γ2⟧ σ1 =
    let ⟨ε1, σ2⟩ = EC⟦γ1⟧ σ1
    in (if success ε1 then ⟨ε1, σ2⟩
        else (let ⟨ε2, σ3⟩ = EC⟦γ2⟧ σ2
              in EC⟦until γ1 do γ2⟧ σ3))
```

Equation 6. The conditional and iterative constructs of the EC valuation function

The final EC equations include the definition of evaluating the empty command ϕ_γ. This is defined for conditional commands in the concrete syntax which require no 'else' branch. The evaluation of a simple command or subshell first demands evaluation of any required I/O redirection by evaluating the ER valuation function.

```
EC⟦φγ⟧ σ1 = SETSTATUS⟦0⟧ σ1
EC⟦α ρ⟧ σ1 =
  let ⟨ε1, σ2⟩ = ER⟦ρ⟧ σ1
  in if success ε1 then let ⟨ε2, σ3⟩ = EA⟦α⟧ σ2
                        in SETSTATUS⟦ε2⟧ σ1
     else SETSTATUS⟦ε1⟧ σ1
EC⟦(γ1) ρ⟧ σ1 =
  let pid = FORK
  in if pid = −1 then ⟨1, PUTERR⟦'cannot fork'⟧ σ1⟩
     else if pid = 0 then
          let ⟨ε1, σ2⟩ = ER⟦ρ⟧ σ1
```

$$\begin{array}{l}\texttt{in if success } \varepsilon_1 \texttt{ then let } \langle \varepsilon_2, \sigma_3 \rangle = \mathrm{EC}[\![\gamma_1]\!]\sigma_2 \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{in EXIT}[\![\varepsilon_2]\!]\sigma_3 \\ \qquad\qquad \texttt{else EXIT}[\![\varepsilon_1]\!]\sigma_2 \\ \qquad\quad \texttt{else SETSTATUS}[\![\texttt{WAIT}[\![\texttt{pid}]\!]]\!]\sigma_1\end{array}$$

Equation 7. The empty, simple and subshell command definitions of EC

The ER valuation function defines I/O redirection in the shell. ER accepts redirection information and a **State** and returns a possibly modified **State** and an exit status indicating if redirection was possible. The returned **State** defines the new input and output streams for the shell after redirection. Here three representative components of the ER function are defined. Other components are similarly defined by calling the SETOUT1 and SETAPPEND1 valuation functions.

$$\begin{array}{l}\mathrm{ER}[\![\phi_\rho]\!]\sigma_1 = \langle 0, \sigma_1 \rangle \\ \mathrm{ER}[\![\langle \alpha\ \rho]\!]\sigma_1 = \\ \quad \texttt{let } \alpha_x = \langle \alpha_0, \alpha_1, \ldots \rangle = \mathrm{EXPAND}[\![\alpha]\!]\sigma_1 \\ \quad \texttt{in if len } \alpha_x = 1 \texttt{ then} \\ \qquad \texttt{let } \langle \varepsilon_1, \sigma_2 \rangle = \mathrm{SETIN2}[\![\alpha_0]\!]\sigma_1 \\ \qquad \texttt{in if success } \varepsilon_1 \texttt{ then ER}[\![\rho]\!]\sigma_2 \\ \qquad\quad \texttt{else } \langle \varepsilon_1, \mathrm{PUTERR}[\![\texttt{'cannot open'} .\alpha_0]\!]\sigma_1 \rangle \\ \qquad \texttt{else } \langle 1, \mathrm{PUTERR}[\![\texttt{'ambiguous < redirection from'} .\alpha]\!]\sigma_1 \rangle \\ \mathrm{ER}[\![\rangle \& \ \alpha\ \rho]\!]\sigma_1 = \\ \quad \texttt{let } \alpha_x = \langle \alpha_0, \alpha_1, \ldots \rangle = \mathrm{EXPAND}[\![\alpha]\!]\sigma_1 \\ \quad \texttt{in if len } \alpha_x = 1 \texttt{ then} \\ \qquad \texttt{let } \langle \varepsilon_1, \sigma_2 \rangle = \mathrm{SETOUT1}[\![\alpha_0]\!]\sigma_1 \\ \qquad \texttt{in if success } \varepsilon_1 \texttt{ then (ER}[\![\rho]\!]\texttt{) SETDUP12}[\![\sigma_2]\!] \\ \qquad\quad \texttt{else } \langle \varepsilon_1, \mathrm{PUTERR}[\![\texttt{'cannot create'} .\alpha_0]\!]\sigma_1 \rangle \\ \qquad \texttt{else } \langle 1, \mathrm{PUTERR}[\![\texttt{'ambiguous}\rangle\texttt{\& redirection to'} .\alpha]\!]\sigma_1 \rangle\end{array}$$

Equation 8. Representative components of the ER valuation function

The valuation function EA either evaluates a *builtin* command, begins reading commands from a new input file (a *shellscript*) or invokes an external program. Which action is performed is determined by the first argument. The 'builtin' commands define actions that must be taken by the shell rather than external programs. These typically include manipulation of the **State** domain. To shorten the function definitions, the following abbreviation is used:

$$\sigma[\![\alpha]\!] ::= \mathrm{GETENV}[\![\alpha]\!]\sigma$$

$$\begin{array}{l}\mathrm{EA}[\![\alpha]\!]\sigma_1 = \texttt{let } \alpha_x = \langle \alpha_0, \alpha_1, \alpha_2, \ldots \alpha_n \rangle = \mathrm{EXPAND}[\![\alpha]\!]\sigma_1 \\ \quad \texttt{in if } \alpha_0 = \texttt{'cd' then} \\ \qquad \texttt{let } \varepsilon_1 = (\texttt{if len } \alpha_x = 1 \texttt{ then CHDIR}[\![\sigma_1[\![\texttt{'HOME'}]\!]]\!] \\ \qquad\qquad\qquad\qquad\qquad \texttt{else CHDIR}[\![\alpha_1]\!]) \\ \qquad \texttt{in if success } \varepsilon_1 \texttt{ then } \langle \varepsilon_1, \sigma_1 \rangle \\ \qquad \texttt{else } \langle \varepsilon_1, \mathrm{PUTERR}[\![\texttt{'cannot cd'}]\!]\sigma_1 \rangle \\ \quad \texttt{elsif } \alpha_0 = \texttt{'exit' then} \\ \qquad \texttt{if len } \alpha_x = 1 \texttt{ then EXIT}[\![0]\!]\sigma_1 \\ \qquad \texttt{else EXIT}[\![\texttt{int} \leftarrow \alpha_1]\!]\sigma_1 \\ \quad \texttt{elsif } \alpha_0 = \texttt{'read' then} \\ \qquad \texttt{if len } \alpha_x = 2 \texttt{ then} \\ \qquad\quad \texttt{let } \langle \alpha_i, \sigma_2 \rangle = \mathrm{READIN2}[\![\sigma_1]\!] \\ \qquad\quad \texttt{in } \langle 0, \sigma_2[(\alpha_i, \mathrm{NoExport})/\alpha_1] \rangle \\ \qquad \texttt{else } \langle 1, \mathrm{PUTERR}[\![\texttt{'usage: read name'}]\!]\sigma_1 \rangle \\ \quad \texttt{elsif } \alpha_0 = \texttt{'set' then} \\ \qquad \texttt{if len } \alpha_x = 3 \texttt{ then } \langle 0, \sigma_1[(\alpha_2, \mathrm{NoExport})/\alpha_1] \rangle \\ \qquad \texttt{else } \langle 1, \mathrm{PUTERR}[\![\texttt{'usage: set name value'}]\!]\sigma_1 \rangle \\ \quad \texttt{elsif } \alpha_0 = \texttt{'setenv' then} \\ \qquad \texttt{if len } \alpha_x = 3 \texttt{ then } \langle 0, \sigma_1[(\alpha_2, \mathrm{Export})/\alpha_1] \rangle \\ \qquad \texttt{else } \langle 1, \mathrm{PUTERR}[\![\texttt{'usage: setenv name value'}]\!]\sigma_1 \rangle \\ \quad \texttt{elsif } \alpha_0 = \texttt{'shift' then} \\ \qquad \texttt{let } \sigma_2 = \sigma_1[(\sigma_1[\![\texttt{'2'}]\!], \mathrm{NoExport})/\texttt{'1'}, \ldots (\sigma_1[\![\texttt{'9'}]\!], \mathrm{NoExport})/\texttt{'8'}] \\ \qquad \texttt{in } \langle 0, \sigma_2[(\texttt{''}, \mathrm{NoExport})/\texttt{'9'}] \rangle \\ \quad \texttt{elsif } \alpha_0 = \texttt{'unset' then} \\ \qquad \texttt{if len } \alpha_x = 2 \texttt{ then else } \langle 0, \sigma_1[(\texttt{''}, \mathrm{NoExport})/\alpha_1] \rangle \\ \qquad \texttt{else } \langle 1, \mathrm{PUTERR}[\![\texttt{'usage: unset name'}]\!]\sigma_1 \rangle \\ \quad \texttt{else EXECUTE}[\![\alpha_0\ \alpha_1\ \alpha_2 \ldots \alpha_n]\!]\sigma_1\end{array}$$

Equation 9. The EA valuation function

A number of auxiliary valuation functions are evaluated by EC, ER and EA. The valuation function SETSTATUS is used to represent the exit status of a command in the **State** domain of the shell. The environment variable referenced by '?' is defined as the string representation of an exit status.

$$\text{SETSTATUS}[\![\varepsilon]\!]\sigma = \langle \varepsilon, \sigma[(\texttt{str} \leftarrow \varepsilon, \texttt{Readonly})/\text{`?'}]\rangle$$

The function PUTERR appends a string to the error output stream of a given **State**.

$$\text{PUTERR}[\![\alpha_0]\!]\sigma_1 = \langle \ldots, \alpha_0.\texttt{out2}, \ldots\rangle \quad \text{where } \langle \ldots, \texttt{out2}, \ldots\rangle = \sigma_1$$

Before any redirection is attempted or any builtin command evaluated or external command executed, function EXPAND is evaluated to map required environment variable values and file names to arguments. Expansion involves replacing environment variable names by their values and performing standard file name expansion. Variable expansion is requested by prefixing a variable name with the '$' character. File name expansion is requested by using metacharacters such as '*' which attempts to match characters in file names. For example:

```
edit $HOME/sem. p* may expand to
  edit/user/chris/sem. paper
```

The PARALLEL valuation function defines asynchronous evaluation of two commands. This evaluation is not performed by the command interpreter itself. Instead, a request is made to the operating system to create a copy of the calling command interpreter process. Each instance of the command interpreter then evaluates a single command. The scheduling of all processes is asynchronous and remains the responsibility of the operating system. The semantic definition of the command interpreter need not include a definition of this scheduling. Methods for formally defining process scheduling in an applicative manner have been presented by Broy.[13]

PARALLEL passes three λ-expressions to the operating system function FORK. FORK first attempts to create a new process to be scheduled. In the event that no more processes may be created (an operating system imposed limit has been reached), the third, error λ-expression is evaluated by the current process and its value returned. If process creation is successful the current process evaluates the first λ-expression and the new process evaluates the second. Each process then returns its result as the result of FORK.

$$\text{PARALLEL}[\![\gamma_1\ \gamma_2]\!]\sigma_1\sigma_2 = \texttt{FORK}(\lambda().\texttt{EXIT}(\text{EC}[\![\gamma_1]\!]\sigma_1),\ \lambda().\text{EC}[\![\gamma_2]\!]\sigma_2,\ \lambda().\langle 1,\ \text{PUTERR}[\![\text{`cannot fork'}]\!]\sigma_2\rangle)$$

Equation 10. The PARALLEL valuation function

3. IMPLEMENTATION IN SML

As demonstrated by Watt,[14] the denotational semantics of significant subsets of programming languages may be implemented and executed directly in metalanguages. As Watt highlights, considerable exposition of denotational semantics is provided by using a functional language as the metalanguage. As a form of validation of our denotational semantics we have implemented the denotational semantic definition of our UNIX command interpreter in Standard ML.

SML is an interactive, statically scoped functional language. Expressions and definitions are presented to the SML interpreter for evaluation. SML evaluates the expression or definition and prints its result along with the most general type information that can be inferred for that result. The strict type checking and type inferencing mechanisms of SML enable type-safe programming without requiring excessive typing syntax which 'clutters' the meaning of a program. This type inferencing is in contrast to the requirements of the MetaIV metalanguage, described in Ref. 3, in which the type of each specification must be explicitly provided.

SML supports abstract data types in which types and interface functions are defined without their implementation being required outside the definition. SML also permits tuples to be passed to and returned from functions. This provides a distinct advantage over the use of a metalanguage such as Pascal in which tuples can only be supported using records and pointers to records.

Although we are defining a command interpreter based on the standard UNIX command interpreters, most, if not all, of our definitions could be implemented on other operating systems. Only six UNIX specific functions are required by our implementation. Each function name is prefixed by the letters 'UNIX'. All of these functions support the creation and control of processes and have equivalents in many other contemporary operating systems. For example, if a host operating system could not support asynchronous execution, the PARALLEL valuation function could be implemented by transparently performing sequential execution. Similarly, the shell's definition of pipes could be transparently supported by using a combination of sequential execution and temporary files created by the shell.

3.1. Implementation of Syntactic Domains

As an introduction to our SML implementation, consider the definition of the shell's syntactic domains. Each syntactic domain is defined using SML's recursive data types. The abstract syntax constructs are defined with value constructor functions as part of their type definition.

```
      type ARG = string;
     type ARGS = ARG list;
datatype RED = nilR
             | readfrom of ARG * RED
             | writeto of ARG * RED
             | appendto of ARG * RED
             | writetoerr of ARG * RED
             | appendtoerr of ARG * RED;
  datatype C = nilC
             | simple of ARGS * RED
             | subshell of C * RED
             | if_cmd of C * C * C
             | while_cmd of C * C
             | until_cmd of C * C
             | semicolon of C * C
             | andand of C * C
```

```
    | oror of C * C
    | background of C * C
    | pipeto of C * C;
```

Program 1. The Syntactic Domains in SML

3.2. Implementation of Semantic Domains

The semantic domains for the shell are defined in SML with abstract type definitions. The representation of each domain and the interface functions are hidden from the rest of the implementation. Access to and modification of instances of each domain are only possible through these interface functions. The definition in SML of the semantic domains is presented below. Ellipsis is used to emphasise that the implementation of the interface functions is hidden from their calling environment.

```
abstype EXIT = state of int
with fun status s = ...
     fun exit s = ...
     fun exits (stat s) = ...
     fun success (stat s) = ...
end;
datatype envtype = Export|NoExport|ReadOnly
abstype STATE = state of (instream*instream*outstream*outstream*
   ((string*string*envtype) list))
with val init_state = ...
     fun putout (str, state state0) = ...
     fun puterr (str, state state0) = ...
     fun setin1 (filenm, state state0) = ...
     (*...other STATE interface functions...*)
     fun setenv (n, v, state state0) = ...
     fun setstatus (exitstatus, state state0) = ...
end;
```

Program 2. Implementation of Semantic Domains with abstract data types

3.3. Implementation of Semantic Equations

The implementation of the valuation functions for the semantic equations is the most important task. As an example of these consider the implementation of some representative EC 'sequential' and 'conditional and iterative' equations. Note the ease with which semantic definitions may be represented in the syntax of SML.

```
fun EC(nilC, state1) = (status0, state1)
  | EC(simple(args, red), state1) =
    let val (ex1, state2) = ER(red, state1)
    in if success ex1 then
       let val (ex2, state3) = EA(args, state2)
       in setstatus(ex2, state1)
       end
       else setstatus(ex1, state1)
    end
|   EC(subshell (c, red), state1) =
   let val pid = UNIXfork ( )
   in if pid = ~1 then (status1, puterr(''Cannot fork subshell'', state1))
     else if pid = 0 then
       let val (ex1, state2) = ER(red, state1)
       in if success ex1 then
          let val (ex2, state3) = EC(c, state2)
          in (exists ex2, state3)
          end
          else (exists ex1, state1)
          end
     else setstatus(status (UNIXwait pid), state1)
    end
```

Program 3. Some representative EC valuation function constructs

```
fun EC(semicolon(left, right), state1) =
    let val (ex1, state2) = EC(left, state1)
    in EC(right, state2)
    end
```

```
| EC(andand(left, right), state1) =
let val (ex1, state2) = EC(left, state1)
in if success ex1 then EC(right, state1)
   else (ex1, state2)
end
```

Program 4. Some sequential EC components in SML

```
fun EC(if_cmd(c1, c2, c3), state1) =
 let val (ex1, state2) = EC(c1, state1)
 in if success ex1 then EC(c2, state2)
    else EC(c3, state2)
 end
 | EC(while_cmd(c1, c2), state1) =
 let val (ex1, state2) = EC(c1, state1)
 in if success ex1 then
    let val (ex2, state3) = EC(c2, state2)
    in EC(while_cmd(c1, c2), state3)
    end
  else (ex1, state2)
end
```

Program 5. Some conditional and iterative EC components in SML

3.4. Implementation of the ER valuation function

The ER valuation function associates indicated files with either input or output streams in a given **State** domain. New instances of the **State** domain are returned by a number of interface functions of the STATE abstract data type. The exit status returned by each of these interface functions indicates if a given file could be opened or created. Here three representative components of the ER function are presented.

```
fun ER(nilR, state 0) = (status 0, state0)
 | ER(readfrom(filenm, red), state0) =
   let val ax = expand([filenm], state0)
   in if length ax = 1 then
      let val (ex1, state1) = setin2 (hd ax, state0)
      in if success ex1 then ER(red, state1)
         else (status 1, puterr(''cannot open''^(hd ax), state0))
      end
   else (status1, puterr(''ambiguous < redirection'', state0))
 end
 | ER(writeoerr(filenm, red), state0) =
   let val ax = expand([filenm], state0)
   in if length ax = 1 then
      let val (ex1, state1) = setout1 (hd ax, state0)
      in if success ex1 then ER(red, setdup12 state1)
         else (status 1, puterr(''cannot create''^(hd ax), state0))
      end
      else (status1, puterr(''ambiguous >& redirection'', state0))
 end
```

Program 6. Some components of the ER valuation function

3.5. Implementation of the EA valuation function

The EA valuation function evaluates a builtin command of the shell, arranges for commands to be read from an indicated file (a shellscript) or executes an external program. The environment variable PATH is obtained from the current **State** and used as list of locations in which to search for a required command. Function `getpath` is used to convert this environment variable to a string list. For example:

$$\text{GETPATH}[\![\text{'PATH'}]\!]\sigma = [\text{'/usr/ucb'}, \text{'/bin'}. \text{'/usr/bin'}] \quad \text{if } \sigma[\![\text{'PATH'}]\!] = \text{'/usr/ucb:/bin:/usr/bin'}$$

The implementation of EA, presented below, is a simple exercise in functional programming.

```
fun EA(args: ARGS, state0) = let val xargs = expand(args, state0)
                                 val nargs = length xargs - 1
                                    val a0 = hd xargs
                                 fun a1( ) = hd(tl xargs)
                                 fun a2( ) = hd(tl(tl xargs))
```

```
in
if a0 = ''exit'' then  (* builtin commands *)
  if nargs = 0 then (exit 0, state0)
  else let val s = makeint(a1( ))
       in (exit(if s> = 0 andalso s < 128 then s else 1), state0)
       end
  else if a0 = ''read'' then
  if nargs = 1 then set(a1( ), readin2 state0, state0)
  else (status 1, puterr(''usage: read name'', state0))
else if a0 = ''setenv'' then
  if nargs = 2 then setenv(a1( ), a2( ), state0)
  else (status 1, puterr(''usage: setenv name value'', state0))
(*... other builtin commands ... *)
else let val pid = UNIXfork( );  (* external commands *)
  in if pid = ~1 then (status 1, puterr(''Cannot fork'', state0))
     else if pid = 0 then
     let fun shellscript arg0 =
       let val (ex, state1) = setin1(arg0, state0)
       in exits(if success ex then shell(setargs(xargs, state1))
                else (puterr(''cannot open''^arg0, state0); ex))
       end
     and tryexec path = status(UNIXexec(path, xargs, state0))
       handle Exec => if access(path, 5) then shellscript path
         else raise Exec
     and trypath nil = raise Exec
       | trypath (h::t) = tryexec(h^''/''^a0)
         handle Exec => trypath t
     in ((if member(''/'',explode a0) then tryexec a0
          else trypath(getpath(''PATH'',state0))), state0)
        handle Exec =>
          (puterr(a0^''not found'', state0); (exit 1, state0))
     end
   else (status (UNIXwait pid), state0)
  end
end
```

Program 7. The EA valuation function

Function `UNIXexec` executes another UNIX program. The first argument indicates the file name of the program to be executed. The new program receives a list of strings as arguments. It also inherits the standard input, output and error streams of the shell and receives all elements of the shell's environment that are tagged for export.

If an indicated file cannot be executed by UNIX but is both readable and executable, as determind by `access`, it is assumed to be a shellscript. The `shell` function is evaluated recursively with a new **State** reflecting that input is to come from this shellscript. The first ten arguments to this shellscript are passed to the shellscript in the environment variables '0' to '9'. Function `setargs` initializes these environment variables. The action of assigning a shellscript's arguments to environment variables in a shell is analogous to the initialization of a function's formal parameters in a programming language.

3.6. Implementation of Input/Output

Input and output is not defined as part of the SML language. Instead, input and output is supported by a library of functions operating on character streams.[15] Two primitive stream types are defined for input and output, named `instream` and `outstream` respectively. With one addition, that of appending characters to a given file, the SML I/O package is sufficient to support our implementation of the UNIX shell.

Each instance of STATE defines the four I/O streams of the shell as I/O streams in SML. Two input streams represent the incoming command stream of the shell and the shell's standard input which is inherited by subprocesses. The distinction is necessary when executing a shellscript – commands are read from an indicated file but the 'read' builtin command reads from the standard input stream. Two output streams represent the standard output and standard error streams which are passed to subprocesses. The initial STATE, `init_state`, contains the shell's initial I/O streams when invoked.

3.7. Supporting an Interactive Interpreter

Denotational semantics of programming languages usually define an initial store for a program consisting of uninitialized or zeroed locations. The initial environment of a program is empty and is modified with declarations in the program. The corresponding **State** domain of a UNIX command interpreter is provided with initial values by the process invoking the interpreter. In our implementation `init_state` contains the initial environment variables exported by the invoking process. Each initial variable is tagged for export by the shell.

Previous executable denotational semantic definitions

have described 'compile-and-execute' programming languages. Such definitions only require a simple parser function accepting a character stream (possibly read from an indicated file) to return an abstract syntax tree representing a program in that language. This abstract tree is then executed by the 'outermost' valuation function. In defining an interactive system such as a command interpreter the parser also requires semantic definition. The parser is a function accepting input from the command stream (initially the terminal), possibly producing error messages announcing syntax errors and finally returning the abstract representation of a **Command**. During evaluation, the parser also requires the environment variables PS1 and PS2 from the current **State**. These variables store the primary and secondary prompt strings and are used when input is required. The parser is thus defined as:

$\texttt{parser} = \lambda(\sigma_1) \rightarrow (\texttt{C} * \sigma_2)$

The I/O streams in σ_1 are used for reading commands, prompting for input and producing syntax errors. In the event that the shell is not interactive (also determined from σ_1), no prompting is performed. The possibly modified σ_2 reflects this I/O.

Being an interactive interpreter there are a number of error conditions which may arise during evaluation from which the interpreter must make a meaningful recovery. Such conditions include the detection of syntax errors in parsing input, the inability of the operating system to create new processes, attempts to modify environment variables tagged as Readonly and the detection of the end of the command stream. SML provides a type-safe mechanism using *exceptions* to indicate such error conditions. Exceptions, when raised in nested functions, may 'percolate' to outer functions. Wherever possible, our implementation does not permit exceptions to escape from the 'inner' valuation functions to the 'outermost' `shell` function. For example, the exception `Syntax-Error` is raised and handled entirely within the `parser` function.

When submitted to the SML compiler the desired type information of the shell's valuation functions is inferred and reported:

```
fun shell state0 =
  let val (cmd, state1) = parser state0
      val (exit1, state2) = EC(cmd, state1)
  in shell state2
      handle io_failure => exits exit1
        | Interrupt => if interactive state2 then shell state2
                       else exits exit1
  end
and sh () = shell init_state;
> val sh = fn: unit -> EXIT
  val shell = fn: STATE -> EXIT
  val EA = fn: (ARGS * STATE) -> (EXIT * STATE)
  val ER = fn: (RED * STATE) -> (EXIT * STATE)
  val EC = fn: (C * STATE) -> (EXIT * STATE)
```

Program 8. The shell and valuation functions and their infered types

3.8. Additions to the Functional Abstract Machine

SML is interpreted by an SECD based interpreter named the Functional Abstract Machine (FAM) [16]. FAM supports rapid evaluation of typeless compiled expressions, leaving the tasks of syntax and type checking to a higher level language such as SML. FAM is defined by its operational semantics of state transitions.

In implementing the denotational semantics of a command interpreter it has been necessary to support a number of functions which cannot be defined directly in SML. These provide the interface between the shell and the operating system. Functions have been provided to invoke new processes, execute programs, open files for appending data, waiting for indicated processes to terminate and to terminate the shell itself. Each of these functions has been added as an instruction in the Functional Abstract Machine by defining its operational semantics. Each new instruction consists of a number of operating system or library calls with parameters and return values consisting of character strings, integers and I/O streams. The new FAM instructions are declared as the following type-safe SML functions:

```
val chdir = fn: string -> bool
val glob = fn: string -> (string list)
val is_term_in = fn: instream -> bool
val is_term_out = fn: outstream -> bool
val open_app = fn: string -> outstream
val UNIXenv = fn: string -> string
val UNIXexec = fn: (string * (string list) * (string list) *
                    instream * outstream -> outstream) -> int
val UNIXexit = fn: int -> int
val UNIXfork = fn: unit -> int
val UNIXpipe = fn: unit -> (instream * outstream)
val UNIXwait = fn: int -> int
```

Program 9. New FAM instructions supported as SML functions

4. CONCLUSION

We have presented a UNIX command interpreter, or shell, defined in terms of its denotational semantics. The shell is both sufficiently complex to demonstrate the usefulness of the denotational semantic definition and to provide practical features. The definition of the shell can easily be extended to provide other accepted shell facilities such as I/O redirection through specified file descriptors and further iterative constructs such as the Bourne's shell's *for* statement.

Direct execution of the denotational semantics is made possible by transcribing the definitions into the meta-language Standard ML. The data abstraction and type inferencing facilities of SML enable implementation of the semantics without requiring excessive typing information. Only in supporting the interface between the command interpreter and operating system was it necessary to extend the facilities of SML. This was performed in a type-safe manner and defined using the operational semantics of the FAM interpreter. Once defined in terms of their denotational semantics, the addition of new command interpreter features is seen simply as an exercise in functional programming.

REFERENCES

1. L. C. Frampton, S. Mellor and C. T. Schlegal, *A Standard Operating System Command and Response Language: The ANSI X3H1 Effort*. In Command Language Directions, Proc. IFIP WG2.7 Working Conference on Command Language Directions, North-Holland (1980).
2. K. Hopper, *User-Oriented Command Language, Requirements and Designs for a Standard Job Control Language*, BCS Monographs in Informatics, Heyden (1981).
3. D. Beech, C. Gram, H. J., Kugler, H. G. Stiegler and C. Unger, *Concepts in User Interfaces: A Reference Model for Command and Response Languages*, Lecture Notes in Computer Science No. 234, Springer-Verlag (1985).
4. T. A. Dolotta and J. R. Mashey, *Using a Command Language as the Primary Programming Tool*. In Command Language Directions, Proc. IFIP WG2.7 Working Conference on Command Language Directions, North-Holland (1980).
5. H. Legard, A. Singer and J. Whiteside, *Directions in Human Factors for Interactive Systems*, Lecture Notes in Computer Science No. 103, Springer-Verlag (1981).
6. J. Madsen, An Experiment in Formal Definition of Operating System Facilities, *Information Processing Letters*, **6** (6), 187–189 (1977).
7. C. Morgan and B. Sufrin, Specification of the UNIX Filing System, *IEEE Trans. on Software Engineering*, **10** (2), 128–142 (1984).
8. J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press (1977).
9. L. Allison, *A Practical Introduction to Denotational Semantics*, Cambridge Computer Science Texts 23 (1986).
10. R. Milner, *The Standard ML Core Language (Revised)*, Edinburgh University (June 1985).
11. M. Gordon, *The Denotational Description of Programming Languages – An Introduction*, Springer-Verlag (1979).
12. A. Giacalone, Doeppner, T. W. Jr. and Braca, M. J. *Toward A Formally Based Programming Environment*, In Integrated Interactive Computing Systems, Proc. European Conference on Integrated Interactive Computing Systems, ECICS 82, North-Holland (1983).
13. M. Broy, Denotational Semantics of Communicating Processes Based on a Language for Applicative Multiprogramming, *Information Processing Letters*, **17**, 29–35 (1983).
14. D. A. Watt, Executable Denotational Semantics, *Software – Practice and Experience*, **16** (1), 13–43 (1986).
15. R. W. Harper, *Standard ML Input/Output*, Edinburgh University (June 1985).
16. L. Cardelli, *The Functional Abstract Machine*, Bell Technical Memorandum TM-83-11271-1, TR-107 (1983).

Announcement

25–28 MARCH 1991

Control '91, Edinburgh

The 1991 International Conference on Control will be held at the Edinburgh Conference Centre, Heriot-Watt University from 25 to 28 March 1991.

The Conference is being organised by the Institution of Electrical Engineers (IEE) and will cover a broad range of topics relating to present trends in the theory and practice of control. The wide scope of the Conference will promote the exchange of information between academics and industrialists in the fields of control, computing, signal processing and instrumentation systems and the following topics.

- Applications within industry
- Artificial intelligence techniques
- Biomedical systems
- Computer-aided control system design
- Computer control systems
- Fault diagnosis
- Manufacturing systems
- H methods
- Large-scale systems
- Man–machine interfaces
- Marine/offshore
- Nuclear
- System simulation
- Measurement and instrumentation
- Nonlinear systems
- Optimisation and numerical methods
- Robotics
- Robust control systems design
- Self-tuning and adaptive control
- System identification and signal processing
- Transport systems
- Aerospace
- Knowledge-based systems
- Networking
- Process control
- Parallel processing and techniques
- Real-time expert control
- Electrical power systems
- Automotive vehicle control
- Environmental systems
- Condition monitoring
- Control and management
- CIM
- Planning and control
- Fault tolerance
- Cybernetics

For press enquiries please contact:

Christina Dagnall, IEE Press Office, Savoy Place, London WC2R 0BL. Tel: 01-240 1871, ext. 272