Commander S — The shell as a browser

Martin Gasbichler Eric Knauel
Universität Tübingen
{gasbichl,knauel}@informatik.uni-tuebingen.de

Abstract

Commander S is a new approach to interactive Unix shells based on interpretation of command output and cursor-oriented terminal programs. The user can easily refer to the output of previous commands when composing new command lines or use interactive viewers to further explore the command results. Commander S is extensible by plug-ins for parsing command output and for viewing command results interactively. The included job control avoids garbling of the terminal by informing the user in a separate widget and running background processes in separate terminals. Commander S is also an interactive front-end to scsh, the Scheme Shell, and it closely integrates Scheme evaluation with command execution. The paper also shows how Commander S employs techniques from object-oriented programming, concurrent programming, and functional programming techniques.

1. Introduction

Common Unix shells such as `tcsh` or `bash` make no effort to understand the output of the commands and built-in commands they execute on the behalf of the user. Instead they simply direct the output to the terminal and force the user to interpret the text own her own. As subsequent commands often build on the output of previous commands, the user needs to enter text that has been output by previous commands. As an example, consider a user that wants to terminate her browser because it hangs once again. She only knows the name of the executable (`netscape`) but not the process ID. Hence she first executes the `ps` command:

```
# ps
  PID     TIME COMMAND
  704  0:00.30 tcsh
 1729  6:01.35 xemacs (xemacs-21.4.17)
 1740  8:10.03 netscape
 5823  0:00.07 tcsh
```

From the output, she learns that the process ID of the browser is 1740. Now she can issue the `kill` command:

```
# kill 1740
```

Even though the previous `ps` command already emitted the process ID 1740, the user has to enter the number manually and double-check to get the right one. Killing processes by name is so common that there is a wide-spread Perl program called `killall` that terminates all running processes with a given name. However, `killall` is not appropriate if multiple processes with the same name exist but only one of them is to be terminated.

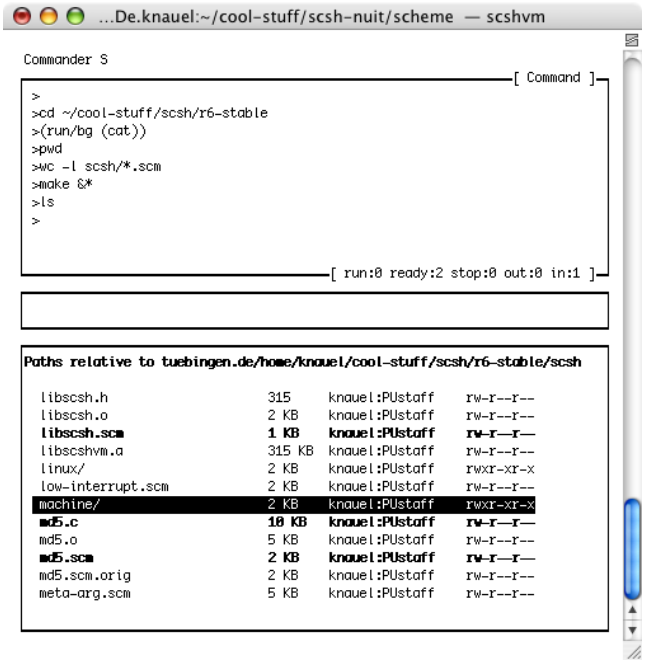

Figure 1. Commander S

Commander S takes a different approach to the concept of an interactive Unix shell: Commander S tries to understand the output of the commands it executes and present it to the user in such a way that the user can easily refer to the output of previous commands. To that end, Commander S draws a user interface on the terminal using the ncurses library. It divides the screen into three areas as shown in Figure 1: The upper half of the screen occupies the *command window* where the user enters the command line. The command line provides the usual line editing facilities such as cursor movement. Below is a small window, called the *current command window*, which shows the last command being executed. The *result window* covers the rest of the screen and contains the output of the last command. The crucial point of the result window is that Commander S presents—for an extensible set of known commands—the result of the commands not simply as text but as structured data. The user can change the focus from the command buffer to the result buffer and *explore* the result. This means that through various key-bindings, the user can invoke other commands that apply to the data presented in the result window. Furthermore, the user can *paste* the data from the result window into the command window to complete the next command line.

Sixth Workshop on Scheme and Functional Programming. September 24, 2005, Tallinn, Estonia.

In the case of the example above, Commander S knows that the result of the `ps` command is a list of processes. It presents this list in the result window as follows:

```
  PID  TIME COMMAND
  704   0:00.30 tcsh
 1729  6:01.35 xemacs (xemacs-21.4.17)
 1740  8:10.03 netscape
 5823  0:00.07 tcsh
```

The result window shows the first line with inverted colors because it is the *focus object*. Some key-bindings modify the focus object only, while others affect the entire result window. Of course, the user can also change the focus object with key strikes. For the list of processes, she needs to press the up and down arrows. To return to the task of killing the browser, user needs to press the down key twice and can then press the key for sending the focus object to the command window. Now, she only needs to add the `kill` command to the command line and press the return key to invoke it. If the user were to kill several processes, she would have to mark them for selection by making one after the other the focus object and pressing the marking key. Then the key for pasting the selection will send them to the command window. Sometimes it is desirable to build the command line not only from the results for the most recent command but from one or more commands that were executed earlier. To support this, Commander S maintains a history for the result buffer in which the user can go backwards and forwards as necessary. This history makes the old results immediately available and the user does not need to use the scrolling facility of the terminal if a command with a larger amount of output happened to be before the result the user is searching for. The current command window always informs the user, which command line produced the output in the result window.

Commander S is also an interactive front-end to scsh, the Scheme Shell. This is realized by a second mode, called *Scheme mode*, for the command window, to which the user can switch from the standard *command mode* with a single key press. The interaction between result window and command window also works for the Scheme mode, but the representation of the pasted objects are s-expressions in this case. The combination of both modes enables the user to combine the power of Scheme with the brevity of shell commands.

In addition, Commander S extends the job control features of common Unix shells. First, the job control facility displays the list of current jobs in the result buffer with key-bindings for the common commands such as putting a job into foreground or background. Second, Commander S uses the ncurses library to continuously display the status of the all current jobs. Finally, Commander S can execute a background job with a separate terminal and allows the user to switch to the terminal, view the running output, or enter new input. To that end, Commander S provides a terminal emulation which stores the output of the process.

1.1 Overview

Section 2 explains some programming techniques and particular libraries used for implementing Commander S. Section 3 gives an overview on Commander S's kernel and describes the implementation of some central features of the user interface. Section 4 describes the interface for writing new viewers. Section 5 pictures some standard viewers such as the process viewer and the directory viewer. Section 6 provides details on the job control implemented by Commander S. Section 7 lists some related work, and Section 8 concludes and presents future work.

2. Preliminaries

This section explains some programming techniques and libraries used to implement Commander S. A reader familiar with the particular techniques may choose to skip the corresponding sections.

2.1 Object-oriented Programming in Scheme

The *viewers* described in Section 5 and Section 4 undertake the task of displaying the result of a command according to its structure. Viewers are implemented in terms of object-oriented programming (Section 4 motivates this design decision). We used the object system proposed by Adams and Rees [2] as a foundation. This system is elegant, easy to implement and very powerful. The complete machinery needed for the object system is given by functions shown in Figure 2. The system represents an object as a procedure that binds the instance variables in its closure and accepts a message (a symbol) as its sole argument. It dispatches on the message and returns the corresponding method as a procedure (see `get-method`). All methods accept the object as their first argument to ensure that overridden methods always get the correct object. Hence, `send`, the construct for calling a method, calls `get-method` first to acquire the actual method and calls that method with the object itself plus the arguments passed to `send`.

2.2 Concurrent Programming using the Concurrent ML API

Commander S is implemented as a concurrent application spawning various threads. To synchronize the threads, Commander S employs a Scheme implementation of the Concurrent ML (CML, for short) concurrency functionality [7]. The implementation is given as a library that is part of *Sunterlib*, the Scheme Untergrund Library [1]. This section provides a short introduction to the subset of the CML API used throughout the implementation of Commander S.

CML offers a collection of data-structures for the communication between threads. For the implementation of Commander S, *synchronous channels* and *placeholders* are important. A channel offers a `send` operation that posts a value to channel and a `receive` operation that reads a value posted to the channel. The communication is synchronous, thus, a `send` operation returns exactly at the time when another thread tries to `receive` a value from the channel (and vice versa). A placeholder is an updateable cell, allowing exactly one assignment. A thread reading the value of a placeholder with `placeholder-value` blocks until another thread updates placeholder with a value using `placeholder-set!`. Updating a placeholder already containing a value yields an error.

The CML frameworks allows the decoupling of describing a synchronous operation from actually performing the operation. Thus, synchronous operations become first-class values, called *rendezvous* in the CML notation. The `receive` operation on a synchronous channel, for example, is composed of generating a rendezvous that describes synchronous operation (e. g. "receive a message on a channel") and waiting till the rendezvous actually occurs. Thus, `receive` is implemented as follows:

```
(define (get-method object message)
  (object message))

(define method? procedure?)

(define (send object message . args)
  (let ((method (get-method object message)))
    (if (method? method)
        (apply method (cons object args))
        (error "No method" message))))
```

Figure 2. Machinery for the object system.

```
(define (receive channel)
  (sync (receive-rv channel)))
```

In which `receive-rv` is a constructor for rendezvous that describe a receive operation on a synchronous channel and `sync` is the function that blocks the thread until a rendezvous actually takes place, or phrased in CML terminology, becomes *enabled*. `Send` and `placeholder-value` may be decomposed in the very same way using `send-rv` and `placeholder-value-rv`.

CML provides combinators that combine multiple rendezvous to a more complex rendezvous. The most important combinator is `choose`, which waits for the first rendezvous from a given list of rendezvous to become enabled. Commander S frequently uses `select`, the synchronous variant of `choose`. The `wrap` combinator allows associating a *post-synchronization action* in form of a function with a rendezvous. When the rendezvous becomes enabled, the associated action is carried out. Function that serve as the post-synchronization actions accept one value — the value that becomes available upon synchronization of the corresponding event. For a receive operation, for example, the value given to the action function is the value received via the channel. The following example code illustrates `select` and `wrap`:

```
(select
 (wrap (receive-rv channel-1)
       (lambda (value)
         (placeholder-set! p value)))
 (wrap (placeholder-value-rv q)
       (lambda (value)
         (send channel-2 value))))
```

Here, `select` combines two rendezvous associated with post-synchronization actions and blocks until the first rendezvous becomes enabled. The first rendezvous in question describes a receive-operation on a synchronous channel named `channel-1`. `Wrap` associates a function with this rendezvous that places the value received via `channel-1` in a placeholder `p`. The second rendezvous describes the synchronous operation of waiting for the value of placeholder `q` becoming available. This rendezvous is also associated with a post-synchronization function which takes the value that just became on-hand and sends it to `channel-2`.

2.3 The ncurses library

Ncurses [3] is a C library that provides a high-level interface to terminal control. In practice a multiplicity of terminal emulations, each having their own control sequences, is in use. Thus, even small tasks like placing the cursor at a certain position on the screen become complex. To assure that an application is portable, the applications needs to know the escape codes of many terminal emulations. Ncurses relieves the programmer of this task. Given a standardized abstract description of a terminal emulation, a so-called *terminfo* entry, usually provided by the maker of the operating system, ncurses learns a particular terminal emulation. The high-level interface of ncurses provides functions for creating overlapping windows, outputting text, controlling the color of output, and placing the cursor. Ncurses also offers a function `wgetch` for reading input from the terminal that decodes the control sequences the terminal emulation uses to encode special keys (such as cursor movement), to a standard representation. We set aside a survey of the ncurses functions used and instead explain their functionality where occur in the following sections.

A separate library for scsh, called `scsh-ncurses`, provides Scheme bindings for all ncurses functions using scsh's foreign function interface. Writing the stubs needed to encode and decode C and Scheme values and calling the ncurses functions is almost straightforward. Just `wgetch` requires special attention. The `wgetch` function reads a character from the terminal, decodes the control sequence if necessary, and returns an integer key code. If no input is available, the behavior of `wgetch` depends on a global mode: in *delay mode*, the function blocks the process until the input becomes available, whereas in *non-delay mode* the functions yields an error. From the perspective of a scsh user, either mode is unfavorable. Calling `wgetch` in delay-mode blocks the whole scsh-process and subsequently all Scheme threads.[1] In non-delay mode, a Scheme thread waiting for input would have to wait busily, thus waste processor time. A preferable mode of operation is to block solely the Scheme thread calling `wgetch`. To achieve this behavior, `scsh-ncurses` calls `wgetch` in non-delay mode at first. If `wgetch` yields an error, `scsh-ncurses` calls scsh's `select` on the terminal to block the Scheme thread calling `select` until the terminal becomes available for reading. Scsh uses the Unix `select` call internally to wait for the file and socket descriptors associated with Scheme ports to become ready for reading and writing. Scsh also offers `select` as Scheme function, which adds the Scheme ports supplied as arguments to the list of file descriptors to watch with the internal `select`.

3. Commander S's kernel

The introduction left unspecified how Commander S recognizes the meaning of a command's output. The idea is not to execute the program directly, but hand over this task to a function that runs the program and parses its output. In the notion of Commander S this function is a *command plug-in*. A command plug-in registers itself as a wrapper for the execution of a certain program. Displaying the parsed output in the result buffer is not in the field of duty of the command plug-in. Instead, *viewer plug-ins* present the output in a structured way. A viewer plug-in registers itself as the presenter for results of a certain type. Command plug-ins are expected to produce a result value of a distinguishable type. Thus, Commander S decouples command evaluation from presentation of the output.

The kernel of Commander S may be regarded as a read-eval-print-loop. Basically, a central event loop processes the input, invokes a command plug-in or executes an external program, and chooses the viewer plug-in to present the result in the result buffer. In Scheme mode, usual Scheme evaluation takes place, but the result is displayed using viewer plug-ins as well. Thus, the evaluation of Scheme expressions also benefits of the power of viewer plug-ins. This section describes the crucial parts of Commander S's kernel.

3.1 Event loop

A central event loop receives all input of the terminal and decides what to do. Basically, the decision depends on two factors: which window has the focus and whether the key pressed has special meaning.

Keys with special meaning, such as the `return` key, are treated by the kernel. The `return` key triggers the evaluation of a command. `Cursor-up` and `page-up` or `cursor-down` and `page-down` keys move through the command history and result history, respectively (see Section 3.6). The key sequence `Control-x` is treated as a prefix, and thus modifies the meaning of the next key press. The sequence `Control-x o` switches the buffer currently focused. `Control-x p` and `Control-x P` paste the current selection and the current focus value, respectively, into the command buffer (see Section 3.3).

If the command window has the focus and the key has no special meaning to the kernel, the key event is passed to the function implementing line-editing (see Section 3.5), which interprets the key accordingly and updates the command buffer. Before continuing in the event loop, the command window needs to be updated to reflect the new state of the command buffer. Thus, the event loop calls a function to repaint the affected part of the command window.

[1] Scsh employs a user-level thread system

⟨command-line⟩ ::= ⟨cmd⟩ (⟨comb⟩ ⟨cmd⟩)* ⟨job⟩?
⟨cmd⟩ ::= ⟨prog⟩ ⟨arg⟩* ⟨redir⟩*
⟨redir⟩ ::= (> | < | >>) ⟨fname⟩
| << ⟨s-expr⟩
⟨comb⟩ ::= | | && | || | ;
⟨job⟩ ::= & | &*
⟨prog⟩ ::= ⟨str⟩ | ⟨unquote⟩
⟨fname⟩ ::= ⟨str⟩ | ⟨unquote⟩
⟨unquote⟩ ::= ,⟨s-expr⟩ | ,@⟨s-expr⟩
⟨str⟩ ::= ⟨scheme-string⟩
| c^+ $c \notin \{$&, |, <, >, ,$\}$

Figure 3. Command language

If the result window has the focus, the key event is passed to the viewer currently visible in the result window. Thus, except for the key sequences listed above, a viewer gets all key events.

3.2 Executing commands

How Commander S executes a command depends on whether the command has been entered in Scheme mode or command mode. If the command buffer is in Scheme mode, the kernel expects the line entered to be a Scheme expression and evaluates it using `eval`. The command mode, in contrast, works akin the prompt of a traditional shell.

The commands entered in the command mode must conform to the *command language* of Commander S. Figure 3 shows a grammar for the command language. Except for some minor differences, this language largely accords to the syntax for commands that users are accustomed to by traditional shells. A notable difference concerns strings, which Commander S models like scsh. While a shell like `tcsh` distinguishes strings in single quotes, double quotes, and backward quotes (for using the output of a command as a string), strings in Commander S's command language are always Scheme strings. The command language is implicitly quasiquoted. Thus, in contexts where a string is expected, the user may use unquote and specify a Scheme expression to be evaluated. Results of the evaluated expression may be a string, a symbol, or an integer. This way, the `tcsh` command `kill ‘cat /var/run/httpd.pid‘` that employs backward quotes to use the contents of the file `/var/run/httpd.pid` as an argument for `kill` may be written as `kill ,(run (cat /var/run/httpd.pid))` in Commander S's command language.

Scsh already supplies a mechanism for running an external program: the `run` macro. This macro expects a specification for the program to run and the redirections of the input and output channels as its arguments. The specification has special syntactic notions called *process forms* and *extended process forms* [9]. Commander S includes a little compiler, which translates a command language command to a process form suitable for the usage with `run`. Thus, when a user submits a command, the compiler generates a corresponding process form and Commander S calls `eval` to actually run the program as specified.

However, the compiled process form demands some preparations before it may be evaluated by `eval`: the `run` macro doesn't substitute shortcuts symbols widely-used by traditional shells. These shortcuts include the tilde, which denotes the user's home directory, environment variable names, and *glob-patterns*. A glob-pattern specifies a list of files by a regular expression. The glob pattern `{/var/tmp,/tmp}/*.scm`, for example, specifies a list of all files with names ending in `.scm` in the directories `/var/tmp` and `/tmp`. Thus, Commander S inserts an expansion pass before evaluating a command that searches the compiled command for shortcuts symbols and replaces them. To implement globbing, Commander S uses the C shell compatible implementation of `glob` that is part of the scsh API.

The evaluation of Scheme expressions takes place in a separate environment called the *shell environment*. The basis for this environment is the module definition of the *shell module* which imports `scheme-with-scsh`, a module providing R^5RS and the whole scsh API. The Scheme 48 module system facilitates turning a module into an environment suitable as an argument for Scheme's `eval` function. Thus, evaluating Scheme expressions boils down to calling `eval` and using the shell environment as the environment for evaluation.

The shell module redefines a choice of scsh functions to return a value with a distinguishable type. `Directory-files` serves as an example; if called without arguments, this function returns the contents of the current working directory as a list of strings. This representation is very handy when writing scripts. However, this representation of directory contents is indistinguishable from an arbitrary list of strings. This poses a problem: the viewer to be used to display a result is selected by examining the result. Thus, the shell module introduces a new record type `fs-object`, which encapsulates a file-system object, and redefines `directory-files` to return a list of `fs-objects`. The redefinition of `directory-files` calls the original definition of `directory-files`, imported with a different name, and wraps the resulting filenames in `fs-object` records. So far the shell module only redefines a few functions that return filenames. An aim of future work is to apply this technique to other parts of the scsh API as well.

3.3 Focus value table

Pasting values into the command window running in Scheme mode requires an external representation of the value. This severely restricts the set of values usable for pasting. For example in scsh records, continuations, and procedures have no external representation. Thus, Commander S allows pasting objects as a reference into a global table called the *focus value table*. View plug-ins may register a value in the table using `add-focus-object` which returns an integer index. The function `focus-value-ref` returns the stored value at a given index. Hence, the viewer plug-in may avoid converting a value to an external representation and return a call to `focus-value-ref` instead.

3.4 Command plug-ins

Command plug-ins undertake the task of running a particular external program, parsing the program's output and representing the result as an distinguishable type. The command plug-in for `ps` exemplifies this. If a user enters `ps` to see the list of running processes, in the command mode of Commander S, this invokes the `ps` plug-in. The `ps` plug-in runs the actual `ps` program provided by the operating system and parses its output. The result is represented as a list of `process` records, thereby making the result distinguishable from an arbitrary list of strings and enabling viewers to recognize the type of the result.

The function `register-plugin!` registers a new plug-in with Commander S. The constructor `make-command-plugin` creates a new command plug-in record which contains three entries: A name for invoking the plug-in, a completion function that calculates completions for the arguments (see Section 3.7), and the *plug-in function*. The kernel calls the plug-in function to run the command, parse the output, and produce the result value. Instead of executing an external program, a plug-in function may also call a scsh function. The following code shows the command plug-in for `printenv` as example. `Printenv` returns a list of all environment variables:

```
(register-plugin!
 (make-command-plugin "printenv"
```

```
no-completer
(lambda (command args)
  (env->alist))))
```

The `no-completer` is a completion functions that offers no completions for a command (see Section 3.7). The scsh function `env->alist` returns all environment variables as an association list.

3.5 Line-editing

The feature users miss most when using scsh in an interactive session is line-editing. Line-editing involves making the backspace key work as expected, allowing the user to move the cursor using the cursor keys, inserting text at an arbitrary position of the command line, and some extra features the user is accustomed to from text editors. The scsh REPL does not provide line-editing because it applies `read` directly to standard input to read from the terminal. However, the command buffer of Commander S offers a line-editing functionality with the features mentioned above and feeds the input into `read` (or the parser for the command language) only after the user has pressed the `return` key. The line-editing functionality is implemented in terms of the ncurses (see Section 2.3), thus is portable and involves no emulation specific code.

3.6 Command and result history

Like conventional shells, Commander S offers a so-called *command history*. A command history provides a way to access the prior commands entered during the session. Most Unix shells bind the cursor keys to a function that cycles through the list of commands and displays prior commands at the prompt. This feature is especially useful when the user executes a series of similar commands.

Besides the command history Commander S also provides a *result history*. The motivation for this novel feature is a limitation of traditional Unix shells that don't provide a method to access the output or result of a prior command execution. In this case the user falls back on a feature of her terminal emulation program. These programs usually buffer the output of the terminal session, thus, the user may scroll up and view the output of commands issued afore. To reuse a prior result the user copies the text to the command prompt using a copy and paste mechanism provided by the terminal program. This method, although exercised by numerous users, has at least two drawbacks. First, it may be hard to find the wanted result — there may be lots of output to search through and the wanted output may even be mingled with another processes output (see Section 6). Second, there is only access to a textual representation of the result.

Commander S saves the result objects created during a session in the result history. Thus, the user may go back in the result history at any time and continue to use a saved result object. The result history facilitates the task of finding the desired result — each command is associated clearly with the result it produced. While cycling through the result history, the active command window shows the command used to produce the result shown in the result buffer.

In the notion of Commander S, a result history is easy to implement. Having the viewer objects (see Section 4) instanced, the kernel stores the object along with the corresponding command in a list that serves as the history. Thus, going back and forth in the history selects an existing viewer object that is set as the the current result object. Subsequently, the kernel clears the result window and sends a `paint` message to the new current result object to make the object visible.

3.7 Programmable completion

Most shells offer an automatic completion for commands and arguments entered partially at the prompt. Usually pressing the tabulator key while editing a command line at the prompt triggers a *completion function*. This function considers the token of the command line the user is currently editing (that is, the token where the cursor is) and finds a set of strings to which the partially entered token is a prefix. This set depicts the set of possible completion for the token. If there is more than one possible completion, most shells simply display the possible completions and expect the user to continue editing the token until the prefix becomes unambiguous. Depending on the position of the token in the command line, the token denotes a program to be executed or an argument to a program. Thus, only executable files come into question as completions for the command token, whereas, intuitively there no such constraint for argument tokens. Most shells accommodate this observation by using different completion functions for the particular tokens of a command line.

Popular shells like `tcsh`, `bash`, and `zsh` offer a *programmable completion function* which allow users to write completion functions tailored to syntax of arguments of a specific command. The file transfer program `ftp`, for example, expects a host name to connect to as its first argument. The following `tcsh` commands establishes an appropriate completion function for `ftp`:

```
> set preferred_ftp_hosts=(ftp.gnu.org ftp.x.org)
> complete ftp 'p/1/\$preferred_ftp_hosts/'
```

This example specifies a completion for the first argument only.[2] The possible completions for this argument are given as a list specified in the variable `preferred_ftp_hosts`.

Commander S provides a similar programmable completion function for the command mode. If the user presses the tabulator key a general completion function calls the parser for the command language and identifies the token the cursor is pointing at. This token is considered for completion. Depending on the position of this token a more specific completion function is selected. The completion function for command tokens is a built into Commander S and uses the union of executables available in the paths listed in `PATH` and the set of registered command plug-in names as possible completions. However, the user may wish to specify an executable by entering a complete path. In this case the command completion function calls `complete-with-filesystem-objects` to build the list of completions. This function checks whether there is a file or directory that matches the partially entered path. If the token matches a directory name, `complete-with-filesystem-objects` offers the contents of this directory as possible completions. Otherwise the parent directory of the partial names is searched for completions.

If a completion function returns a single possible completion, Commander S may replace the token on the command line with this completion and repaint the command prompt. However, if there is more than one completion, Commander S uses the result buffer to display the list of completions. The user may use the list as an aide to memory and continue to type the token, or by pressing the tabulator key a second time switch the buffer focus and select a completion using the cursor keys directly.

The general completion functions also handles the completion of arguments. Unless a specific a completion function for the current command token is specified, it calls `complete-with-filesystem-objects` to complete the argument. Specific completion functions are tied to command plug-ins. Thus, to provide a special completion function, the user adds a command plug-in. The following command plug-in for the `ftp` command provides such a completion function.

[2] `p/1` stands for "position one"

```
(register-plugin!
 (make-command-plugin
  "ftp"
  (let* ((hosts '("ftp.gnu.org" "ftp.x.org"))
         (cs (make-completion-set hosts)))
    (lambda (command to-complete)
      (completions-for
       cs (or (to-complete-prefix to-complete) ""))))
  just-run-in-foreground))
```

In this example, the second argument to `make-command-plugin` is the completion function. A completion function has two arguments; the abstract syntax of the command line and the token to completed. The completion function in question uses a built-in list of host names as possible completions. `Make-completion-set` creates a special caching data-structure which speeds up the computation of matching completions. This is especially useful when the set of possible completion is big, for example, when searching the completions for file names. The procedure `completions-for` searches and returns the matching completions for the prefix returned by `to-complete-prefix` in the completion set.

4. Implementing Viewer Plug-ins

In the notion of Commander S a *viewer plug-in* (*viewer* for short) undertakes the task of displaying the result value of a command in a structured fashion. However, a viewer may go beyond just displaying data and implement a small application running in the result window. The predefined file system viewer (see Section 5), for example, not only displays files and directories but also allows navigating through subdirectories.

Given a result value, Commander S tries to find the appropriate viewer. Each viewer comes with a predicate that identifies the result values the viewer handles. Commander S applies the predicates provided by the registered viewers to the result value. The viewer belonging to the first predicate to evaluate to true accepts the bid. Now, Commander S instances a new viewer using the accordant constructor and asks the viewer to paint itself to the result window.

Viewers are implemented using object-oriented programming (see Section 2.1 for an introduction of the object system used). A viewer depicts an object that accepts the messages sent by kernel and encapsulates a state. In this setting, an object-oriented approach appeared to be a natural choice. Commander S sends the following messages to viewer objects:

- `paint` The `paint` message asks the object to paint itself to the result window. This message is sent to objects just created or if an result object becomes the current result object. (i.e., if the user cycles through the result history, see Section 3.6). As arguments, the objects receives the ncurses window to paint in, a result buffer object which contains information about the result window's size, and a boolean indicating whether the result window has the focus.
- `key-press` If the result window has the focus, the current result object receives a `key-press` message whenever the user presses a key. The object receives the key code and a boolean saying whether the special prefix key sequence `Control+x` is active as arguments. The kernel expects this method to return an instance of the viewer and stores this instance in the history. This is a clincher, since this allows a viewer to instantiate and return a different viewer. The viewer responsible for displaying the contents of a user record, for example, uses this case to instance a directory viewer object if the user presses return key on the line displaying the path to a user's home directory.
- `get-selection-as-ref` This message asks the viewer to return the current selection as a reference into `focus-value-table` (see Section 3.3) received as an argument. The message is only available if the command buffer is in Scheme mode, thus, the return value of this method has to be a piece of Scheme code (as a string).
- `get-selection-as-text` This message asks the viewer to return the current selection in a textual representation. If selections don't make sense in context of a result value, this method may return false. A boolean delivered as an argument says whether the selection is to be inserted into the command or Scheme mode. Thus, a viewer may deliver an adequate string (see Section 4.2 for an example). It is conceivable though that representing the selection as a string makes no sense. In this case a viewer may choose to understand the `get-selection-as-text` message as a `get-selection-as-ref` message, hence, requiring a reference to the `focus-value-table`. To facilitate this, the `focus-value-table` is passed as a second argument to the `get-selection-as-text` messages.

4.1 Selection lists

Before giving an example for the implementation of a viewer object, we shall describe *selection lists*. Selection lists are an important user interface widget, akin to menus, used by almost all viewer objects. A selection list displays a given set of entries as sequential lines at an arbitrary position inside an ncurses window. Using the cursor keys, the user may move a selection bar over the lines to focus a particular entry, and mark and unmark entries. Most viewers employ a selection list using marking to facilitate selecting items which are to be processed together. The selection list also determines the area in view if the number of items to display exceeds the space assigned to the selection list.

The constructor `make-selection-list` expects as its argument a list of records of type `element` that denote the items of the selection list, and returns a Scheme record representing the selection list. An `element` record consists of a field that carries the object to be returned if the user marks the accordant line, a boolean saying whether this entry may be marked at all, and the text to be displayed.

The `paint-selection-list-at` operation accepts a selection list, window-based coordinates, and an ncurses window as its arguments and paints the selection list in its current state at the given coordinates to the window. To pass key events to a selection list, viewer objects call the function `selection-list-handle-key-press` which updates and returns the state of selection list accordingly.

Implementing a `get-selection-as-text` method in a viewer frequently boils down to getting the list of marked entries from a selection list using `selection-list-get-selection`, or, if this list is empty because no entries are marked, getting the entry currently focused by the selection bar using `selection-list-selected-entry`. The selection list implementation offers the function `make-get-selection-as-ref-method` which returns a function suitable as an implementation of a `get-selection-as-ref` method. The focus objects returned by methods implemented using this function stand for the return object specified in the accordant `element` record.

4.2 Example: process viewer

As an example for the implementation of viewers, this section describes the implementation of the process viewer from the introduction and sketches the implementation of the command plug-in for `ps`.

The process viewer views the output of the `ps` command. The `ps` command is a command plug-in based on the `portable ps` library from Sunterlib [1]. As the `ps` command is not standardized, the library dispatches on the type of the host operating system and then issues the `ps` command with options chosen to get all processes

and a set of additional information available on all supported platforms. It then parses the output and stuffs it info a record of type `process`. The `ps` command plug-in does not currently support additional options but returns this list unchanged. In the future, the `ps` command should accept arguments to restrict the returned processes and customize the additional information. While argument parsing is certainly more work, a user who often switches operating systems would certainly be happy to use the same set of options on all platforms. Of course, the syntax of the options could easily be made customizable.

Figure 4 contains the implementation of the viewer plug-in for processes. The function `make-process-viewer` is the constructor for process viewer objects. The constructor is called by the kernel, if the predicate for this viewer, `list-of-processes?`, identified a result value as a list of process objects. The kernel supplies the result value in question and the buffer to draw to as arguments to the constructor. The constructor returns a function that given a message name returns a function implementing the method. The instance variables of the object are bound in the closure of this function. The process viewer employs a selection list (see Section 4.1) to display a list of processes. `Make-process-selection-list` formats the process objects and uses `make-selection-list` to create a selection list that fits into the result window leaving one line free for a heading. On a `paint` message, the viewer displays the header and calls the procedure `paint-selection-list-at` to draw the selection list beneath the header. A `key-press` message is also forwarded to the selection list. On a `get-selection-as-text` message, it returns the PIDs of the selected processes for the command mode and a list of PIDs in the Scheme mode.

Finally, the last two lines of the figure register the process viewer plug-in registers as viewer for a list of records of type `process` and hands out the constructor to the kernel.

5. Predefined viewers

The previous section already presented Commander S's viewer for processes. In this section, we present further viewers for filesystem objects, user and group information and results of commands related to AFS. In addition, a viewer for inspecting arbitrary Scheme values is described.

5.1 The filesystem viewer

Dealing with files is another common scenario where the user is forced to re-enter text that appeared in the output of a previous command. A common pattern is that the user first issues an `ls` command to list the files within a directory and then uses another command to manipulate certain files. To view the most recent error log file of an Apache web-server, the user could first use `ls -lat`, which prints the files sorted by date:

```
# ls -lt
-rw-r--r--   5543 Jun 15 02:00 error_log.1118275200
drwx--x---    512 Jun 15 02:00 ./
-rw-r--r--  49024 Jun 14 15:04 access_log.1118275200
-rw-r--r--  66312 Jun  8 21:59 access_log.1117670400
-rw-r--r--  11498 Jun  8 21:59 error_log.1117670400
-rw-r--r-- 140048 Jun  1 18:17 access_log.1117065600
-rw-r--r--   4688 Jun  1 05:36 error_log.1117065600
drwx--x---    512 Mar 25  2004 ../
```

Next, she would invoke a viewer such as `less` on the latest file `error_log.1118275200`:

```
# less error_log.1118275200
```

Again, the user has to enter text that appeared in the output of a previous command. Modern shells such as `bash` or `tcsh` will help the user to enter by providing *command line completion*. This means that the shell examines the command line already typed and completes the last token as far as possible or presents the user a set of possible completions. The shell derives the possible completions from the leading command, the default mode is to complete the token as a filename. In the example above, the user could ask the shell to complete the command line `less e`. The shell will expand this to `less error_log.111` and list all error files as possible completions. Now the user needs to inspect the output of the previous `ls -lt` command to learn that the name of the most recent file continues with an `8`. After entering this character, the shell is able to fully complete the filename. However, while command line completion is certainly of great aid for the programmer, the shell again makes no use of the output of previous commands, which contains in our example the files in chronological order. If the example takes place within the `tcsh` shell, this is especially disappointing as there `ls` is a built-in command. This means, the output is not produced by some external command but by the shell itself.

The user could try to save typing by combining entering a command line that extracts the name of the newest error log for the output of `ls` and calls `less` on it:

```
(define (make-process-viewer processes buffer)
  (let* ((processes processes)
         (cols (result-buffer-num-cols buffer))
         (lines (result-buffer-num-lines buffer))
         (sel-list
          (make-process-selection-list
           cols (- lines 1) processes))
         (header (make-header-line cols)))

    (define (get-selection-as-text
               self for-scheme-mode?
               focus-object-table)
      (let* ((marked
              (selection-list-get-selection sel-list)))
        (cond
         ((null? marked)
          (number->string
           (process-info-pid
            (selection-list-selected-entry sel-list))))
         (for-scheme-mode?
          (string-append
           "'" (exp->string
                (map process-info-pid marked))))
         (else
          (string-join
           (map process-info-pid marked))))))

    (lambda (message)
      (case message
       ((paint)
        (lambda (self win buffer have-focus?)
          (mvwaddstr win 0 0 header)
          (paint-selection-list-at
           sel-list 0 1 win buffer have-focus?)))
       ((key-press)
        (lambda (self key control-x-pressed?)
          (set! sel-list
                (selection-list-handle-key-press
                 sel-list key))
          self))
       ((get-selection-as-text) get-selection-as-text)
       ((get-selection-as-ref)
        (make-get-selection-as-ref-method sel-list))
       (else
        (error "process-viewer unknown message"))))))

(register-plugin!
 (make-viewer make-process-viewer list-of-processes?))
```

Figure 4. Implementation of the process viewer (excerpt).

```
# less `ls -1t err* | head -n 1`
```

While this approach is close in the spirit of the Unix philosophy to combine little tools to perform the work, the command line is rather long and fragile. We would not dare to use such a construction on the command-line for a command such as `rm`. It also requires the user to know in advance that error logs (and only these) start with `err`.

Commander S knows that the result of the `ls -lat` command is a list of files. It presents this list in the result window as follows:

```
Paths relative to /usr/local/svn/logs

-rw-r--r--   5543 Jun 15 02:00 error_log.1118275200
drwx--x---    512 Jun 15 02:00 ./
-rw-r--r--  49024 Jun 14 15:04 access_log.1118275200
-rw-r--r--  66312 Jun  8 21:59 access_log.1117670400
-rw-r--r--  11498 Jun  8 21:59 error_log.1117670400
-rw-r--r-- 140048 Jun  1 18:17 access_log.1117065600
-rw-r--r--   4688 Jun  1 05:36 error_log.1117065600
drwx--x---    512 Mar 25  2004 ../
```

That is, the presentation of a list of files is the list of the file names relative to a directory, which is displayed in the first line. If the focus object is a directory and the user presses the return key, the result window will display the contents of this directory. To return to the task of viewing the latest log file, the user can immediately press the key for sending the focus object to the command window, as the focus object is already the most recent file. Now, she only needs to add the `less` command to the command line and press the return key to invoke it. Pasting files to the command window inserts them as absolute filenames. If the command window is in Scheme mode, pasting inserts filenames as strings.

If the user enters the `ls` command, Commander S does not really invoke the `ls` program and parse its output. Instead, it uses the scsh function *file-info* to obtain the file status information and the function `directory-files` to get the contents of a directory. From this information, it generates a list of records of type `fs-object`. An `fs-object` combines a filename with file status information. The filesystem viewer registers itself as the viewer for `fs-objects` and for lists of `fs-objects`.

As Commander S provides its own binding for the scsh procedure `directory-files`, which returns a list of `fs-objects` instead of a list of strings, and extends the scsh functions which operate on filenames to `fs-objects`, the viewer is also able to present the values of Scheme expressions returning lists of filenames.

The functionality of filesystem viewer could be extended in various aspects: additional key-bindings for renaming, deleting, or copying files, manipulation of file mode bits, invoking of a default application based on the filename suffix, and so forth. However, while we would certainly like to have these features, it is not the focus of our current work as programs like midnight commander or the dired plug-in for Emacs already show the merits of this approach. Instead, Commander S aims combine graphical presentation with command execution and shell programming. Unlike pure front-ends for filesystem browsers, Commander S is also not limited to the presentation of filesystem objects.

5.2 User and group information viewer

User and group information are ubiquitous in Unix. For user information, scsh provides the procedure `user-info` as wrapper for the standard C functions `getpwnam`/`getpwuid` to return the user information from a given login name or UID. It returns a record `user-info` with the fields `name`, `uid`, `gid`, `home-dir`, and `shell` which contain the corresponding entries form the user database (usually `/etc/passwd`). For the group information, scsh analogously provides a wrapper `group-info` for the C functions `getgrnam`/`getgrgid`. The fields of the returned record `group-info` are `name`, `gid`, and `members`, the latter containing the users of the group as a list of strings. Commander S contains viewers for the `user-info` and `group-info` records that present the contents of the records in a selection list. The main feature of these viewers is that the user may navigate through the presented information by selecting an entry and pressing the return key: For the `gid` field, Commander S presents the corresponding group information, for the `home-dir`, it invokes the filesystem viewer from Section 5.1 on the home directory, likewise for the `shell` field, and for the members of a group, Commander S presents the associated user information. Here is an example for the value of the expression `(user-info "gasbichl")`:

```
[0: name] gasbichl
 [uid] 666
 [gid] 4711
 [home-dir] /afs/wsi/home/gasbichl
 [shell] /bin/tcsh
```

If the user presses the return key, Commander S presents the information for GID 4711 as follows:

```
 [name] PUstaff
 [gid] 4711
 members:
 gasbichl
 klaeren
 knauel
```

The viewers are implemented in about 130 lines of code but already provide a nice tool for browsing user and group information. We think that in this style a lot of information in the realm of Unix can be presented and thus enable the user to browse this information very conveniently and fast.

5.3 AFS

This section presents two viewers related to the Andrew File System (AFS for short) as an example for using Commander S for viewing the result of special purpose programs. AFS is a network filesystem based on a client-server model. AFS stores the data on the server in logical partitions called *volumes*. Each volume is mounted at some directory below the global `/afs` root. On the client, a local daemon transparently fetches and stores the contents of the volumes from the server and maps it into the local filesystem. AFS also introduces permissions for directories based on access control list (acl for short) and has its own user management. The user views the permissions with the `fs listacl` command and manipulates them with the `fs setacl` command. For example:

```
# fs listacl .
Access list for . is
Normal rights:
  system:administrators rlidwka
  gasbichl rlidwka
  knauel rl
# fs setacl . knauel rli
```

adds the right to insert files into the current directory for the user `knauel`. Commander S saves the user from entering the username that already occurred in the output by displaying the result of `fs listacl` using a selection list:

```
Access list for . is
Normal rights:
system:administrators rlidwka
gasbichl rlidwka
knauel rl
```

By pressing the key for sending the selection, the user can paste the string `knauel rl` to the command window running in command mode behind a `fs setacl`. Alternatively, the user may paste the entry as a pair while in Scheme mode. This is especially useful to set the rights of several users at once. For example, the following expression grants the right to read, list and insert files to a list of such entries which the user would paste from the result window at the place of `...`:

```
(for-each (lambda (acl)
            (fs setacl "." (car acl) "rli")) ...)
```

On the other hand, the viewer for `fs listacl` also supports direct editing of the acl entries. Currently, pressing the deletion key removes an entry from the acl. More features such as direct modification of the rights would be desirable but requires functionality beyond the current capabilities of the selection list.

Commander S also supports management of volumes. The command `fs listquota` takes as argument a directory and prints the quota information for the volume the directory resides in. This is also a convenient way to obtain the name of the volume needed the most volume-related commands. Commander S prints the result of `fs listquota` as

```
Volume Name: home.gasbichl
Quota:       1000000
Used         899724
% Used       90%
Partition    28%
```

From here, the user can either paste the volume name into the command window or press the return key to execute the `vos examine` command on the volume. A future version will also support direct editing of the quota.

The commands for volume manipulation also have command line completion for the volume name argument. Commander S receives the list of all volumes from the command `vos listvldb`. Executing this command may take some time, therefore it is not desirable to initialize this list during startup. Fortunately, command completion is completely programmable in Scheme and during startup the corresponding plug-in can simply spawn a thread which issues `vos listvldb` and initializes the volume list. This way, the user has to wait only if she wants command completion for `vos` before the thread finishes its work.

5.4 Value inspector

The domain of viewers is not limited to the results of Unix commands. In fact, the user may add viewers for any kind of Scheme value. Scheme 48 already comes with an inspection facility to browse arbitrary Scheme values. We have lifted the inspection facility into our ncurses-based framework and use it as the default viewer for exceptions which effectively implements a debugger.

We briefly review the inspection facility in Scheme 48: Its command processor provides a command `,inspect` that takes as its argument a Scheme expression, evaluates it and presents the outermost structure of the resulting value in a menu. There is a menu entry for every immediate sub-value. For a list, the sub-values are the entries of the list, for a record the sub-values are the components of the record, for a continuation the contents of the stack frame makes up the sub-values. A menu entry consists of a number for selection by the user, an optional name for reference, and the external representation of the sub-value. The source of the name depends on the kind of value being inspected: for records it is the name of the record field, for environment frames it is the name of the variables. List or vector entries do not have names. After the presentation of the menu, the user may enter the number of a menu entry to continue inspection with the corresponding sub-value or press the `u` key return form the inspection of a sub-value. For continuations, the `d` key selects the parent continuation. If there are more than 14 sub-values, the `m` key switches the presentation of the menu to the next 14 sub-values and so on. Finally, the `q` key ends the inspector and sets the focus object of the command processor to the last value that has been inspected. The command processor also comes with a `,debug` command which inspects the continuation of the last exception that occurred. As inspection of a continuation displays an excerpt of the source code of the corresponding function call before presenting the menu, this is enough to implement a very useful debugger.

For Commander S we implemented a viewer, called *inspector*, which shows the sub-values of an arbitrary Scheme value in a selection list. The user may select a sub-value by moving the selection bar to it and pressing the return key. In addition, we have adopted the key-bindings for `u` and `d` from Scheme 48.

For the implementation of the inspector, Commander S mainly reverts to the procedure `prepare-menu` from the implementation of the `,inspect` command. The procedure takes as its argument a Scheme value and returns the list of its sub-values as pairs of a name (or `#f`) and the sub-value. Commander S turns these pairs into `element` records for a selection list: The object to be returned on marking is the sub-value itself, all elements are markable, and the text is the external representation shortened to the width of the window. For the latter, we make use of `limited-write`, another utility from Scheme 48 which is a variant of `write` that limits the output to a certain depth and output length. Unfortunately, the single line within a selection list of often not enough space to present complex data structures in a useful manner. Besides the preparation of the selection list, there is not much to do for the inspector: As the `,inspect` command, it prints a source code excerpt for continuation in a header line and being able to return from a sub-value requires the viewer to maintain a stack of visited values. Invoking the inspector on a sub-value pushes the current value on the stack and the `u` key pops a value from the stack and makes it the current value. The `,inspect` command in Scheme 48 proceeds likewise.

We could use the inspector to display any value but we have currently only registered it for the continuations of exceptions, but this may be extended for arbitrary values.

6. Job control

Most Unix shells allow the user to run multiple processes simultaneously. In shell terminology these processes are called *jobs*. A shell usually provides commands to stop and continue jobs, view the list of jobs and their status, and the job's access rights to the terminal. All processes share a single terminal as their standard output and input. The POSIX job control interface [5] enables the shell to control which process may read or write to a terminal. Traditional shells pursue the following policy: A single foreground job has read and write access to the terminal and all background jobs are allowed write to the terminal only. If a background job tries to read from the terminal, the shell suspends the execution of the job until the job becomes the foreground job.

Thus, running multiple background jobs, which write to the terminal yield a mingled output. Basically, the user has two choices to avoid this: redirecting the output of each job to a separate file, or make the shell's job control stopping processes that attempt to write to the terminal. However, both options are disadvantageous. A job control policy with exclusive write access may stop the computation of a background job completely just because there is output available. This not appropriate in all cases, for example when running a daemon from the command line. On the other hand, redirecting the output requires extra effort for setting up the

redirections for standard output and standard error, viewing the file, and deleting the temporary files afterwards.

Commander S adds a third method, not provided by traditional shells, to the picture; so-called *console jobs*. The standard input and output of a console job are connected via a separate pseudo terminal to Commander S. A thread continuously reads the pseudo terminal to ensure that writing to the terminal does not block. A *console* record stores the pseudo terminals and the buffered output of a job. The viewer plug-in for this record type displays the output of the job in the result buffer and updates it continuously as new output arrives. Thus, the user may review the output of a command at any time. Section 6.3 discusses console jobs in detail and presents the implementation at a glance.

Beside console jobs, Commander S offers job control as known from traditional shells. The implementation, however, diverges from traditional implementations. We present a elegant concurrent implementation in the CML framework in the following sections.

Section 6.1 presents the POSIX job control facilities at a glance. A reader familiar with these facilities and their mode of operation may choose to skip this section. Section 6.2 describes how Commander S runs jobs without a separate console. Section 6.3 explains the execution of console jobs. Section 6.4 describes the implementation of the job list, a data structure that maintains the informations on jobs centrally.

6.1 Traditional job control

The POSIX API contains functions for implementing job control which are widely-used by traditional shells. Scsh already provides bindings to these functions. Thus, it was not necessary extend scsh to implement Commander S's job control. This section explains the basics of POSIX job control using scsh's names for the POSIX functions.

Process groups are the basis for job control — a process group is a set of processes, which share a common process group id. Each process is member of exactly one process group. When a process forks, the child process inherits the process group id of the parent — the process is said to *join* the parent's process group. A process may also *open* a new process group by calling `set-process-group`. Each terminal device is associated with one process group, named the *foreground process group*, all other process groups are called *background process groups*. A process group makes itself to the foreground process by calling `set-tty-process-group`. In contrast to processes of background process groups, processes of the foreground process group are granted read and write access to the terminal. If a background process tries to read from the terminal, the kernel terminal driver suspends the job using the `SIGTTIN` signal. Depending on the configuration of the terminal a background job writing to the terminal may also be suspended using the `SIGTTOU` signal. Using `wait`, a parent process may watch if a child gets suspended.

6.2 Jobs without console

Jobs without a separate console are either foreground or background jobs and work akin to jobs in a traditional shell. To execute a foreground job, Commander S temporarily escapes the curses mode and hands the control on the screen over to the foreground job. Once the foreground jobs terminates (or gets suspended by a signal), Commander S reobtains control. Commander S expects a background job neither to read from nor write to the terminal. If the job tries to read or write, however, the job gets suspended and Commander S notifies the user (see Section 6.4). In this case the user may choose to continue the job in foreground. Vice versa, a user may also explicitly stop a foreground job and continue the job in background.

```
(define-syntax run/fg
  (syntax-rules ()
    ((_ epf)
     (run/fg* '(exec-epf epf)))))

(define (run/fg* s-expr)
  (debug-message "run/fg* " s-expr)
  (save-tty-excursion
   (current-input-port)
   (lambda ()
     (def-prog-mode)
     (clear)
     (endwin)
     (restore-initial-tty-info! (current-input-port))
     (drain-tty (current-output-port))
     (obtain-lock paint-lock)
     (let ((foreground-pgrp
            (tty-process-group (current-output-port)))
           (proc
            (fork
             (lambda ()
               (set-process-group (pid) (pid))
               (set-tty-process-group
                 (current-output-port) (pid))
               (eval-shell-env s-expr)))))
       (let* ((job (make-job-sans-console s-expr proc))
              (status (job-status job)))
         (set-tty-process-group
           (current-output-port) foreground-pgrp)
         (newline)
         (display "Press any key to return...")
         (wait-for-key)
         (release-lock paint-lock)
         job)))))
```

Figure 5. Running a job in foreground.

The machinery for running jobs is built on top of scsh's `run` form. The form `(run/fg` *`epf`*`)` executes the extended process form *epf* as a foreground job. To specify a program to run and the corresponding redirections of the input and output channels scsh uses a special syntactic notation: process forms and extended process forms. Thus, `run` and `run/fg` are implemented as macros not as functions.

Figure 5 shows the implementation of `run/fg`. Applications of `run/fg` expand into a call to `run/fg*`; a function that expects a piece of Scheme code as a s-expression as its argument. The Scheme code is supposed to actually run the process using scsh's basic `exec-epf` facility. Unlike `run`, `exec-epf` does not fork the process before running the program. `Run/fg*` calls `eval-shell-env` to evaluate the Scheme code in the shell environment. It is important that the evaluation takes place in the shell environment since an extended process form is implicitly backquoted, thus, by using unquote, a user may embed Scheme code in an extended process form. Carrying out the evaluation in the shell environment ensures, for example, that the user may refer to variables defined interactively in the Scheme mode or use focus values.

Before running the process using `eval-shell-env`, `run/fg*` calls a sequence of ncurses functions to save the current screen, clear it and finally escapes the curses mode temporarily using `endwin`. This yields an empty screen called the *result screen*. This avoids that the Commander S screen is garbled with the output of the process. To execute the process, `run/fg*` forks the process, opens a new process group, and makes this process group the new foreground process group. The parent process calls `make-job-sans-console` to create a new job record with the process object returned by `fork`. The parent process uses `job-status`; a wrapper version of `wait` for jobs. Thus, the parent waits until

the child process exits and makes itself the foreground process group again. Afterwards, the parent process waits for a key press to give the user time to read the child's output. It is essential to ensure that no output occurs during the time Commander S is a background process — otherwise the terminal driver would suspend Commander S. To enforce this condition `run/bf` obtains the `paint-lock` which prevents other threads, such as the thread that updates the job status indicator (see Section 6.4), from painting onto the screen.

Running jobs in background works alike using a function `run/bg*`. There, the code for escaping from the curses mode and setting the foreground process drops out. On start-up, Commander S configures the terminal to stop background processes that try to write to terminal, thus, a background cannot garble the screen. Commander S offers two functions for continuing suspended jobs without a console: `Continue/fg` puts a stopped job into the foreground and continues the job, `continue/bg`, vice versa, continues a job as a background job. The implementation of this functions is derived from the implementation of `run/fg*` and `run/bg*`. However, instead of forking and calling `exec-epf`, the functions send the process group of the job a `SIGCONT` signal, thus, the processes continue to execute.

6.3 Console jobs

The implementation of console jobs is more complex than the implementation of jobs without console. While there is no extra effort needed to display the output of job without console — it is only visible on the separate result screen — the output of console jobs causes more effort. The output of a job must be read by Commander S continuously to keep the job running. However, displaying the output in the result buffer as it occurs is not reasonable — the job would behave like an ordinary foreground job.

Here, the concept of viewer plug-ins comes into play. The output of a console job is represented by a *console* record. An accompanying viewer plug-in for this record type displays the output and updates the result buffer as new output arrives. To the kernel a console is conceptually just another value with a predefined viewer plug-in. Each console is accompanied by a thread that reads the pseudo terminal of the process and sends the characters read into a synchronous CML channel. Thus, this thread lifts I/O events into the CML framework.

To actually paint the contents of the output buffer to the screen, the console viewer plug-in uses a so-called *terminal buffer*. The heart of a terminal buffer is a thread spawned by the function shown in Figure 6. The terminal buffer is connected via the synchronous `pty-channel` to the thread that reads the console's output. Depending on whether the console is currently visible in the result buffer or not, the terminal buffer either buffers the new output (by calling `terminal-buffer-add-char`) or buffers it and immediately repaints the result buffer. The decision whether to update the result buffer or not is left up to the console viewer plug-in, which uses `resume-console-output` or `pause-console-output` to stop and continue the updates, respectively.

The terminal buffer performs a second task hidden in the function `terminal-buffer-add-char`. Basically, this function implements a terminal emulator for a small subset of VT100 control codes. The terminal emulation is necessary to restrict the effects of terminal escape codes generated by the running job to the result buffer only. Forwarding the escape codes rawly to terminal Commander S is running on yields undesirable effects. If the running job outputs the escape code to clear the screen, for example, this escape code would be interpreted by the terminal emulator for the terminal Commander S is running on, and clean the entire screen— including the command buffer. Alas ncurses offers no solution to this problem.

```
(define (spawn-console-loop
         pause-channel resume-channel
         window termbuf pty-channel)
  (spawn (lambda ()
    (let lp ((paint? #t))
      (select
       (wrap (receive-rv pause-channel)
             (lambda (ignore)
               (lp #f)))
       (wrap (receive-rv resume-channel)
             (lambda (ignore)
               (lp #t)))
       (wrap (receive-rv pty-channel)
             (lambda (char)
               (cond
                ((eof-object? char) (lp paint?))
                (else
                 (terminal-buffer-add-char
                  termbuf char)
                 (if paint?
                   (begin
                     (curses-paint-terminal-buffer
                      termbuf window)
                     (wrefresh window)))
                 (lp paint?)))))))))))

(define (pause-console-output console)
  (send (console-pause-channel console) 'ignore))

(define (resume-console-output console)
  (send (console-resume-channel console) 'ignore))
```

Figure 6. Updating a `terminal-buffer` and painting it.

6.4 Job status and job list

A job is in one of the following run states: running, finished, stopped, waiting for input, or waiting with output (the latter applies to background jobs without a console only). Traditional shells notify the user either immediately or before drawing the next prompt if the status of a job changes. Both methods have drawbacks: a prompt notification means that the shell prints the notification directly to the terminal at point of time the status change occurs, thus garbling the terminal output. Waiting for the next prompt avoids a garbled screen, but the user has to issue (empty) commands from time to time to see if a status change occurred. A graphical user interface produces relief for this problem.

Commander S's command buffer displays a small gauge, the *job status indicator*, in the lower right corner of the command window (see figure 1). The job status indicator displays the current number of processes in each of the possible state. Whenever the status of a jobs changes, a thread updates the job counts immediately without disrupting the user.

Commander S uses a central *job list* to maintain a list of all jobs. The job list serves two purposes. First of all, it is needed to implement the `jobs` command, which prints a list of all jobs and their current state. As a second task, the job list registers all status changes of a job and informs the job status indicator about the change.

The implementation of the job list was tricky — there are several sources of events that modify the state of the job list: A user may submit a new job at the prompt, stop or continue a job, or a background job may interrupt or finish its execution. Thus, the job list needs to observe several diverse sources for events at once. First of all, user commands such as submitting, continuing, or stopping a job need to inform the job list about the job status changes. The termination or suspension of a background jobs is the second source for events that trigger changes in the state of

the job list. To notice these changes the job list needs to call `wait` for each background job and update the job list. Using the CML framework these diverse sources for events may easily be represented uniformly as rendezvous. Thus, one central `select` synchronously waits for the occurrence of any of the named events.

Figure 7 shows an excerpt from the implementation of the job list. The function `spawn-joblist-surveillant` starts the thread that maintains the job list and returns the `statistics-channel`. This channel connects the job list with the job status indicator — whenever the state of the job list changes in a relevant way, the job list posts the updated job counts to this channel and the thread accompanying the job status indicator updates the gage. The thread spawned by `spawn-joblist-surveillant` executes an infinite loop that uses `select` to choose a rendezvous from the possible sources of events affecting the job list. The job list consists of lists for each run state that are bound locally in the thread. The loop variable `notify?` indicates whether an update of the job status indicator is due. If this is the case, the thread sends the current job counts to `statistics-channel`. The constructor for jobs `make-job-sans-console` and `make-job-with-console` submit the jobs just created to job list using the `add-job-channel`. If a rendezvous on the `add-job-channel` is enabled, the function associated to this event by the `wrap` combinator adds the new job to the list of running jobs and continues the loop. In this case an update of the job status indicator is due, thus the loop function is called with `#t` as the value for `notify?`. Receive rendezvous on the `notify-continue/foreground-channel` indicate that the user issued a `continue/fg` or `continue/bg` command. Thus, a job that is either stopped, waiting with output, or waiting for input changes to the running state. The accordant action for this events deletes the job from the lists for stopped jobs, adds it to the list of running job, and sets `notify?` to true. The `get-job-list-channel` is used by the `jobs` command to get the list of all jobs.

The job list also monitors the status changes of the processes using `wait`. The constructor for jobs spawns a thread that calls `wait` on a job's process object, and fills a CML placeholder with the status value returned by `wait`. The function `job-status-rv` returns the corresponding rendezvous. This way, the status change of a process translates to a CML rendezvous suitable for integration with the job list's `select` call. Thus, the job list surveillance thread includes the `job-status-rv` for all running jobs into the selection of rendezvous by mapping `job-status-rv` on the list of all running jobs. The function associated with each rendezvous adds and removes the affected job to the corresponding lists of jobs in a specific state. The scsh functions `status:exit-val`, `status:stop-sig`, and `status:term-sig` decode the `status` value returned by `job-status-rv`. Depending on whether the process exited, was suspended or terminated abnormally, these functions return `#f` or an integer providing further information on the reason of state change. If the operating system suspend the process, for example, `status:stop-sig` returns the signal number that yielded to suspension.

7. Related Work

There is multiplicity of file managers available that follow the tradition of the abandoned Norton Commander, such as the GNU MidnightCommander [6] or LFM [8]. These applications use most of the screen to display one or two file lists which the user may navigate, use to select files, and perform operations on them. The last line of the screen shows the shell prompt of a traditional shell. Thus, these applications are clearly committed to work with files solely. To Commander S, working with files is just one facet of a more holistic approach for easing the work with a shell. The GNU MidnightCommander comes with job control for background jobs but these "jobs" are merely running copying and moving operations. XEmacs and GNU Emacs ship with `dired`, a special mode for editing directory trees [10]. The GNU screen [4] terminal manager allows users to detach from a terminal and reattach to it later, and offers some text based copy and paste mechanism. This provides a functionality akin to Commander S's console jobs.

8. Conclusion and Future Work

This paper presented Commander S as a browser for UNIX. With the aid of command plug-ins, Commander S parses the output of commands and acquires the contained information. Viewer plug-ins use the ncurses library to present the output information as interactive content. Commander S contains plug-ins for the most common entities in shell interaction, processes, and filesystem contents. The paper shows that it is possible with little effort to extend Commander S to other domains. Through the use of the CML library, the implementation of the job control is very short, even though it is more powerful than in common UNIX shells and even contains a small terminal emulator for running processes in the background while saving their output.

The technique presented in this paper could be used to present other information such as DNS result records, or the contents of NIS or LDAP databases. As Commander S closely integrates an evaluator for Scheme expressions, the user can always fall back to writing small programs if the power of the command language or the viewers does not suffice to accomplish a task.

One conceivable extension of Commander S is the integration with the Orion window manager which is also based on Scsh. In this combination, Orion would start several Commander S instances concurrently, and assign every instance its own pseudo terminal and Xterm window.

Acknowledgments Christoph de Mattia wrote the `scsh-ncurses` bindings and an early prototype of Commander S called `scsh-nuit`.

References

[1] Sunterlib — the Scheme Untergrund library, 2005. Available at `http://www.scsh.net/resources/sunterlib.html`.

[2] Norman Adams and Jonathan Rees. Object-oriented programming in Scheme. In *ACM Conference on Lisp and Functional Programming*, pages 277–288, Snowbird, Utah, 1988. ACM Press.

[3] Eric Raymond, Zeyd Ben-Halim, and Thomas Dickey. *Writing programs with ncurses*, 2004.

[4] Oliver Laumann et al. *GNU Screen 4.0.2 user manual*, 2005. `https://savannah.gnu.org/projects/screen/`.

[5] Donald. A Lewine. *POSIX Programmer's Guide*. O'Reilly & Associates, Inc., 1994.

[6] Pavel Roskin and Miguel de Icaza. The GNU MidnightCommander, 2005. `http://www.ibiblio.org/mc/`.

[7] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[8] Iñigo Serna. lfm —last file manager, 2004. `http://www.terra.es/personal7/inigoserna/lfm/`.

[9] Olin Shivers, Brian D. Carlstrom, Martin Gasbichler, and Mike Sperber. *Scsh Reference Manual*, 2003. Available from `http://www.scsh.net/`.

[10] Michael Sperber. Dired. `http://www-pu.informatik.uni-tuebingen.de/users/sperber/software/dired/%`.

```
(define (spawn-joblist-surveillant)
  (let ((statistics-channel (make-channel)))
     (spawn (lambda ()
       (let lp ((running '()) (ready '()) (stopped '()) (new-output '())
                (waiting-for-input '()) (notify? #f))
         (cond
          (notify?
           (send statistics-channel ...)
           (lp running ready stopped new-output waiting-for-input #f))
          (else
           (apply select
            (append
             (list
              (wrap (receive-rv add-job-channel)
                    (lambda (new-job)
                      (lp (cons new-job running)
                          ready stopped new-output waiting-for-input #t)))
              (wrap (receive-rv notify-continue/foreground-channel)
                    (lambda (job)
                      (lp (cons job running) ready
                          (delete job stopped) (delete job new-output)
                          (delete job waiting-for-input) #t)))
              (wrap (receive-rv get-job-list-channel)
                    (lambda (answer-channel)
                      (send answer-channel ...)
                      (lp running ready stopped new-output waiting-for-input #f))))
             (map
              (lambda (job)
                (wrap (job-status-rv job)
                      (lambda (status)
                        (cond
                         ((status:exit-val status)
                          => (lambda (ignore)
                               (lp (delete job running) (cons job ready) stopped
                                   new-output waiting-for-input #t)))
                         ((status:stop-sig status)
                          => (lambda (signal)
                               (cond
                                ((= signal signal/ttin)
                                 (lp (delete job running) ready stopped new-output
                                     (cons job waiting-for-input) #t))
                                ((= signal signal/ttou)
                                 (lp (delete job running) ready stopped
                                     (cons job new-output) waiting-for-input #t))
                                ((= signal signal/tstp)
                                 (stop-job job)
                                 (lp (delete job running) ready (cons job stopped)
                                     new-output waiting-for-input #t))
                                (else (error "Unhandled signal" signal)))))
                         ((status:term-sig status)
                          => (lambda (signal)
                               (lp (delete job running) ready (cons job stopped)
                                   new-output waiting-for-input #t)))))))
              running))))))))
    statistics-channel))
```

Figure 7. Excerpt from the implementation of a job list with asynchronous status indication.