Caml-Shcaml *

An OCaml Library for Unix Shell Programming

Alec Heller   Jesse A. Tov

Northeastern University

{alec,tov}@ccs.neu.edu

Abstract

Objective Caml is a flexible, expressive programming language, but for manipulating Unix processes, nothing beats the terseness and clarity of Bourne Shell:

```
ls *.docx | wc -l
```

To achieve the same effect in C requires several more lines of code and, very likely, a glance at the manual page for *readdir*(). Despite OCaml's excellent `Unix` module, which provides access to a wide variety of system calls, a task as simple as counting the `.docx` files in the current directory is hardly easier in OCaml than in C. The code looks largely the same.

Caml-Shcaml addresses this problem by bringing high-level abstractions for Unix systems and shell programming to OCaml. In particular, we take advantage of OCaml's type system to offer statically-checked data pipelines. External Unix processes and internal OCaml stream transducers communicate seamlessly within these pipelines. Shcaml brings other essential systems concepts into the world of typed functional programming as well, including high-level interfaces to Unix facilities for I/O redirection, signal handling, program execution, and subprocess reaping.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming—Objective Caml; D.4.9 [*Operating Systems*]: Systems Programs and Utilities — Command and control languages; D.4.1 [*Operating Systems*]: Process Management

General Terms Languages

Keywords Shell programming, Objective Caml, Unix, domain-specific languages, types

* This work was generously supported by Jane Street Capital.

This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *Proc. ACM SIGPLAN Workshop on ML,* 9:79–90, 2008. http://doi.acm.org/10.1145/1411304.1411316

ML'08, September 21, 2008, Victoria, BC, Canada.
Copyright © 2008 ACM 978-1-60558-062-3/08/09. . . $5.00

1. Introduction

> Master Foo once said to a visiting programmer: "There is more Unix-nature in one line of shell script than there is in ten thousand lines of C."
>
> Rootless Root: The Unix Koans of Master Foo

The functional programmer and the shell programmer view the universe differently. The former is most comfortable with typed, structured data; the latter is accustomed to swimming in streams of bytes. The similarity between composing functions and piping processes together begs to be exploited. In practice, unfortunately, these two paradigms don't play so nicely together.

Many language libraries provide ways to request system services. Traditionally, these have comprised wrappers around low-level system calls or higher-level calls such as *system*(), which is specified to "be as if a child process were created using *fork*(), and the child process invoked the sh utility using *execl*()" (POSIX 2004). That is, to run a command (represented as a string) by passing it off to the system shell to interpret: `system("cut -d: /etc/passwd ...")`.

This is not enough. It ought to be as easy as possible to move data back and forth without writing a lot of boilerplate. To use the low-level interface (see Figure 5 for an example), a programmer must write the sort of control structures and error handling that makes traditional systems programming unpleasant and error-prone. Using *system*() is little better, providing so little control over the child process that only the most unsophisticated tasks can reasonably be executed.

We intend Shcaml to be a middle path bridging these two views of systems programming. It provides both detailed control and elegant abstraction over the Unix environment. Shcaml achieves this through an API and operators that are strongly reminiscent of the Bourne Shell (Bourne 1978) without losing their essential 'OCaml flavor.' While Shcaml makes heavy use of OCaml's type system, structured data, and other functional language features, it retains the Unix nature.

The working thesis of Shcaml's design is that by enabling the seamless flow of data between OCaml code and external processes, programmers may mix and match the styles of computation that

```
cat /etc/passwd                          |
  cut -d: -f1,7                          |
  grep ':/usr/bin/scsh$'                 |
  sed 's/^\(.*\):/\
    echo "Check out Caml-Shcaml\!" | \
    mail -s "A new shell" \1/'           |
  sh
```

Figure 1. The problem

```
run begin
  from_file "/etc/passwd"                     -|
  Adaptor.Passwd.fitting ()                   -|
  grep (fun line -> Passwd.shell line =
                            "/usr/bin/scsh") -|
  cut Passwd.name                             -|
  mail_to ~subject:"A New Shell"
          ~msg:"Check out Caml-Shcaml!"
end
```

Figure 2. The Shcaml solution

they feel are most appropriate to solve their tasks. It is not difficult to write a single, ad-hoc function to perform a task more easily done by a shell utility. However, we believe that a well-designed library will allow programmers to code at the same level of abstraction as the shell, but within a larger functional program.

1.1 A Better Shell Script

The somewhat onerous shell program in Figure 1 sends mail to every user on the system whose login shell is `/usr/bin/scsh` to notify them of the presence of Shcaml. It achieves this by consulting the `/etc/passwd` file, where users are associated with (amongst other things) their login shell. Data in the `passwd` file is stored in colon-separated seven-tuples, with one record per line.

Because we are only interested in usernames and login shells, we generate a data stream containing each username associated with its shell. The `cat` command reads the `passwd` file and sends it along the pipeline to the `cut` invocation. In turn, `cut` projects the first and seventh fields of each password record in the `passwd` file, splitting on `:`, giving us a stream of *user*:*shell* pairs. Then, we use `grep` to filter out non-Scsh users. We use `sed` next to transform each *user*:*shell* pair into a shell command to send the desired email, and pipe that to `sh` in order to evaluate it.

While it may appear somewhat obscure, this program is nonetheless a direct and natural implementation. The comparable program written in Shcaml will share much of the structure of the shell script. Like in the shell, Shcaml pipelines manipulate streams of data. Our data streams (which we call *shtreams*) extend OCaml's standard stream abstraction with several new features. Shtreams may be represented not only as a generating function, but explicitly as an input channel and reader function. This facilitates passing a shtream directly as input to another external Unix process. We discuss shtreams in more detail in § 3. In our example, we wish to generate a shtream of data from the `passwd` file, so we write

```
from_file "/etc/passwd"
```

for the first component in the Shcaml version of the pipeline.

If we want to test the `shell` field of the data that `grep` is filtering, we must first be able to find the field in each line. In our example, we must know that the shell field is the seventh component of the record. Shcaml provides facilities for abstracting this information, so that we may use meaningful names rather than field numbers. We provide these operations through *adaptors*, which we discuss further in § 5.1. To parse the information in `passwd` file lines into records and add it to the data in our shtream, we use the provided `Passwd` adaptor:

```
Adaptor.Passwd.fitting ()
```

Shcaml provides a variety of other adaptors, each of which contains information for parsing a particular file format.

The Shcaml version of `grep` simply filters the lines of data it receives by the given predicate:

```
grep (fun line -> Passwd.shell line = "/usr/bin/scsh")
```

We begin to see some of the power of Shcaml in this code. The function `Passwd.shell` projects out the `shell` component of a shtream containing data from the `passwd` file. Despite the apparent simplicity, `Passwd.shell` is part of Shcaml's more structured analog to lines of text in normal Unix pipelines. The type of `Passwd.shell` ensures that any shtream of data which flows through it *must* have a `shell` field to project. Because it is common to write pipelines that add and remove different kinds of information in a data stream, Shcaml *lines* support operations that OCaml's record, tuple, and object types do not.

Once we have only those lines corresponding to Scsh users, we may extract their usernames to send our mail message. First we project their usernames:

```
cut Passwd.name
```

Finally, we pass the usernames to a function that sends the mail message.

To put all these pieces together, we require something like Unix's pipes. Shcaml provides a combinator library for fitting shtream computations such as the above into larger pipelines. Functions from shtream to shtream can be lifted into *fittings*, which can then be piped together with other fittings to create pipelines. We discuss fittings in § 5. Once we have a fitting that represents our whole computation, we may *run* that fitting, which causes the computation actually to be executed. Our complete Shcaml rendition of the script may be found in Figure 2.

While both the shell version and Shcaml version are written as pipelines, only the shell version has multiple Unix process communicating over Unix pipes (Figure 3(a)). By contrast, The Shcaml version comprises several stream transducers that communicate by passing OCaml values within a single process (Figure 3(b)).

This code demonstrates the style in which Shcaml programs are written, but it does not yet demonstrate the integration of external Unix programs with OCaml code. Suppose that instead of using the actual `passwd` file, we are given a shell script `get-passwd.sh` that synthesizes a large amount of `passwd`-formatted data from a directory service. We need only replace the first fitting, `from_file "/etc/passwd"`, with

```
command "get-passwd.sh"
```

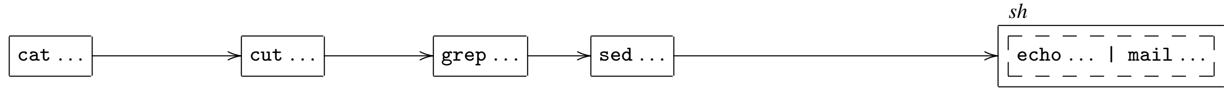
This replaces the first stage in the pipeline with an external process, which communicates over a Unix pipe with the second stage, which is written in OCaml (Figure 3(c)).

Now, suppose that some users have requested not to receive such announcements. Another script, `remove-opt-out.sh`, will filter out their usernames. So we may add a call to this script in the appropriate spot in the pipeline:

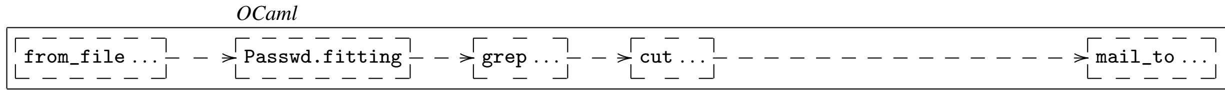
```
run begin
  command "get-passwd.sh"                     -|
  Adaptor.Passwd.fitting ()                   -|
  grep (fun line -> Passwd.shell line =
                            "/usr/bin/scsh") -|
  cut Passwd.name                             -|
  command "remove-opt-out.sh"                 -|
  mail_to ~subject:"A New Shell"
          ~msg:"Check out Caml-Shcaml!"
end
```

In this case, we have inserted an external process in the middle of the pipeline, and it communicates transparently with the OCaml processes on each side (Figure 3(d)).

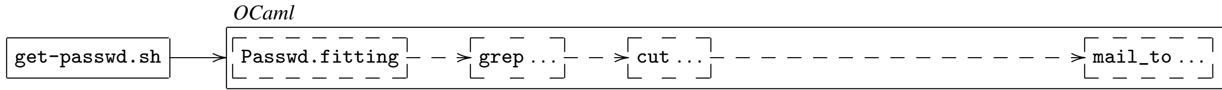
We may continue this process, swapping OCaml code for external programs, or vice versa, as we desire. We need only be mindful that external processes receive an untyped string version of the data in each record, and produce untyped bytes that must be parsed again if we are to make sense of them.



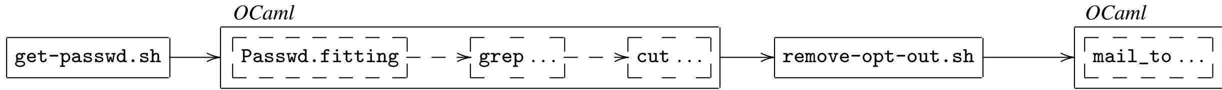
(a) Several UNIX processes started from a shell script.



(b) A Shcaml pipeline within one OCaml process.



(c) Shcaml with one external UNIX process and one OCaml process.



(d) Shcaml with two external UNIX processes and two OCaml processes.

Figure 3. Examples of pipelines

2. Background

In most programming language implementations on Unix, it is not terribly challenging to execute subprocesses. However, to use these subprocesses with the facility of a shell script requires delicate and error-prone resource management. Additionally, Unix processes communicate through character streams. Thus, even when programming in a high-level language with sophisticated features for the description and manipulation of data, dealing with other Unix programs requires constant parsing and unparsing of data as it passes to and from other processes. If we wish to take advantage of shell-style abstractions in our functional programs, we need a library that provides the shell's key abstractions—redirection, pipes, and foreground and background processes—at that higher level, rather than at the level of system calls.

Modern scripting languages such as Perl and Ruby are often considered appropriate tools for coordinating shell tasks with general purpose computation. However, their approach to integrating with Unix is no more sophisticated than most other languages. Perl and Ruby both provide some convenient syntax for simple shell interactions, but what makes these languages especially effective is more likely their string matching features. String handling is not nearly so easy in OCaml as it is in Perl, and reading string data from an external process is often a less pleasant experience. Because working with structured data is better than working with collections of strings, Shcaml mitigates these shortcomings of OCaml with respect to other scripting languages by handling a large proportion of the string parsing automatically.

Perhaps the most well known project to bring shell programming into the functional realm is Shivers's (1994) Scheme Shell, Scsh. Indeed, the design and implementation of Shcaml has been strongly influenced by it. Scsh defines an embedded domain-specific language for process creation and I/O redirection, known as *process forms*. These are expanded by various special forms, which execute the specified shell actions in different ways such as in the foreground, the background, or by replacing the current process image. This feature of Scsh is implemented using macros; Shcaml, on the other hand, uses a combinator language to construct fittings—first-class OCaml values—which are then interpreted by functions known as *runners*.

Recognizing that "the string is a stark data structure and everywhere it is passed there is much duplication of process" (Perlis 1982), Scsh provides tools to ease the burden of dealing with string data. While character streams are the *lingua franca* of Unix, newline-delimited records are the most common dialect. A wide variety of Unix programs consume and produce line-oriented data formats. In Scsh, then, the idea is that an incoming character stream will first be split into strings representing records (often at newlines, but not necessarily), then each record string will be split into fields and given to the user in a list. This is a vast improvement over dealing directly with a stream of characters, but in Shcaml, we take this idea another step: a stream of OCaml `string lists` is still rather stark, and we prefer to work with records having more precise types. This enables a natural means of computing on the data produced by external processes from within a typed language.

Shcaml distinguishes itself from Scsh most clearly in two intertwined features: its use of combinators to create pipelines and its use of the type system to express the structure of data flowing along those pipelines. Unlike Scsh's process forms, which are defined in terms of macros, Shcaml's pipelines are first-class OCaml values. Furthermore, because the types of these pipelines control what data must be present in the stream flowing through them, we obtain stronger guarantees about the sanity of the pipeline computation.

Shcaml is not the first attempt at a tight integration between OCaml and the shell. Verlyck (2002) implemented Cash, a reasonably direct port of Scsh to OCaml.[1] While Cash provides high-level facilities for managing processes and creating pipelines, it provides nothing analogous to Scsh's process forms, and its data model does not take advantage of OCaml's type system.

Windows PowerShell, formerly Monad Shell, is Microsoft's new interactive shell and automation language (Snover 2002). PowerShell scripts are composed of *cmdlet*s, which are .NET classes that implement various shell and management facilities such as removing files or listing processes. Cmdlets may be connected into object pipelines, which transmit streams of .NET objects, rather than text as in Unix. As in Shcaml, this allows PowerShell pipelines to deal in rich data, and it avoids much of the trouble and poten-

[1] So direct, in fact, that the module names in Cash include section numbers from the Scsh manual.

tial inaccuracy of parsing. Unlike Shcaml, PowerShell scripts are untyped and make heavy use of reflection. Hotwire Shell (Walters 2007) is a program similar to PowerShell, but with a greater emphasis on graphical interface concerns.

3. Shtreams

Shtreams are the central abstraction in Shcaml's tool kit. In Unix, data is communicated between processes in a pipeline, which transmits streams of bytes. This enables filter-style programs in Unix to be relatively modular, because many communicate textual data via the same interface. In Shcaml, we want to delegate parts of our computation to other programs, but we also may want to write other parts directly in OCaml. Those parts should communicate among themselves with OCaml values rather than untyped bytes. Shcaml uses shtreams to give a common interface to both *internal* generators that produce OCaml values and *external* processes connected over Unix pipes. In many cases, this makes them interchangeable.

Regardless of whether a particular shtream's data source is internal or external, the Shcaml `Shtream` library provides operations to treat shtreams as stream-like or channel-like. The stream-like interface is modeled on OCaml's `Stream` module. Some operations include

```
next : 'a Shtream.t -> 'a
```

which produces the next element of the given shtream (or raises an exception), and

```
map : ('a -> 'b) -> 'a t -> 'b t
```

which lazily maps a function over a shtream. In order to use a shtream as input for an external process, we need to turn the shtream into an input channel:[2]

```
channel_of : ?procref:Proc.t option ref ->
             ?before:(unit -> unit) ->
             ?after:(unit -> unit) ->
             ('a -> unit) -> 'a t -> in_channel
```

If given an external shtream, which produces elements by reading from a channel, `Shtream.channel_of` merely returns the underlying `in_channel`; it ignores the other four arguments. If the given shtream is internal, however, then `channel_of` forks a new process that it connects to the current process by a Unix pipe. The new process then evaluates the rest of the shtream, using the `'a -> unit` function to output each element. The optional arguments `before` and `after` are thunks that the new process forces before and after evaluating the shtream, in case the data format it produces needs a file header or footer. Finally, when `channel_of` needs to fork, it stores the process ID of the child process in `procrec`.

Shtreams may also be constructed from either internal or external data sources. As in OCaml's `Stream` module, a shtream may be constructed from a generating function, such as

```
from  : (int -> 'a option) -> 'a Shtream.t
```

which calls the generating function once, with an integer index, for each element as the shtream is forced. We can also construct an external shtream from any input channel:

```
of_channel : ?hint:(Reader.raw_line -> 'a) ->
             (in_channel -> 'a) -> in_channel -> 'a t
```

Given a function of type `in_channel -> 'a` that reads items from a channel, `Shtream.of_channel` constructs the shtream of those items. We discuss the optional argument `hint` in § 5.1.

[2] In OCaml, a formal argument `?label:type` indicates an optional keyword argument `label` with type `type`.

Unix command	Stream combinator
`grep`	*filter*
`sed s///`	*map*
`xargs`	*apply*
`sh`	*eval*
`cut`	*map* π_i
`paste`	*zip*

Table 1. Common Unix commands and their stream combinator counterparts

This design has an important implication: When two external processes are composed, they speak directly over a Unix pipe, with no OCaml code shuttling or marshalling data. When two internal Shcaml transducers are composed, they run in the same OCaml process, with no Unix pipe or marshalling in between. But when an external process is composed with an internal stream, they are able to communicate over a pipe, with marshalling happening on the OCaml end.

3.1 Shtream Implementation

Shtreams are represented as an algebraic datatype with several constructors:

```
type 'a data =
  | Strict of 'a * 'a data
  | Delay  of (unit -> 'a data)
  | Extern of in_channel
  | TheEnd of Proc.status
```

The constructors `Strict` and `Delay` are used to represent internal shtreams: `Strict` represents shtreams whose head has already been evaluated, and `Delay` represents a totally unevaluted shtream. The constructor `Extern` represents an external shtream, whose contents are read from the `in_channel`. Finally, `TheEnd` represents an exhausted internal or external shtream, including the exit status.[3]

A shtream's `'a data` is actually wrapped up in a record having several other fields that influence the shtream's behavior.

```
type 'a shtream = {
  mutable data: 'a data;
  (* ··· *)
}
```

4. Lines

Many of the most common and well-known shell utilities have analogs among the standard stream combinators. We list some of these correspondences in Table 1. Notably, all of these programs are *line-oriented*, in the sense that they process their input as newline-delimited streams of strings. Often, each line corresponds to a record in some tabular data format. Shcaml pipelines are also essentially line-oriented. However, rather than streams of strings, they compute with shtreams of record values called *line record*s.

In principle, a line of text flowing through a Unix pipeline represents a record of data. Each program in the pipeline interprets the line in some way, possibly parsing it into the relevant internal representation, and then serializes its output records back into the stream. In our prototypical example, the `passwd` file is composed of lines, each a record having seven components separated by the colon character. A program that is computing with data from the `passwd` file will almost certainly split the file into lines, and often

[3] It is possible for a shtream to begin with some number of internal `Strict`s and `Delay`s, and subsequently become external. This might happen if, for example, a finite internal shtream is appended to an external shtream.

split each line into a more structured record with convenient access to the seven fields.

We believe there are several problems with this approach:

- Parsing is often approximate, as data formats are not always sufficiently well defined so that we can be sure that one pipeline component's printer agrees with the next format's parser.
- The need to serialize each record as easily parsable text limits the quantity and variety of metadata that may be easily attached to each record.
- In quick, ad-hoc coding, it is often not worth the trouble of designing a correct parser and easy-to-use representation. When that one-shot script resurfaces in a year, it may not be clear which field `$4` represents.

It would be better if stream transducers that operate on rich data could communicate language values where the type system can see them, rather than marshalling at each step.

Given that, consider the shell pipeline

```
cat /etc/passwd | cut -d: -f1
```

which produces the usernames of all users in the `passwd` file. If we replace both commands with internal stream operators, what types should they have? The first component, `cat`, is producing a stream of lines from the `passwd` file, so it should produce seven-element records with names corresponding to the fields in `/etc/passwd`. Thus, `cut` must consume those records and produce usernames.

Clearly, if `cut` is to be generally useful, it must be polymorphic. A `cut` that was good only for extracting string fields from password records would not be of much use. Rather, `cut` should take as its argument a line record selector whose domain matches the line records that `cut` is receiving in its input. For this to be maximally flexible, the selector itself should be polymorphic as well, in that it only cares that the line records have the field that it projects—it is irrelevant what other fields may be present. For example, were we to annotate each password record with information about the user's quota, last login, and mail spool, a function that looks only at the username should be oblivious to the other data. We contend that polymorphic record types, or row types, are an ideal solution.

4.1 Row Types

Wand (1987) introduced row types as a means for inferring object types in a typed, object-oriented language. In a language with row types, operations on records may be polymorphic in the fields of the records on which they operate. For example, in a hypothetical row-type calculus, we may express the notion of a record containing a string-typed field labeled user and *some other fields*:

$$\langle \mathsf{user} : string; \rho \rangle.$$

The type variable ρ is an *extension variable*, and it stands for the other fields that might be present in a record having this type.

Consider, for example, a function that takes a record r and selects fields $\mathsf{shell} : string$ and $\mathsf{user} : string$ from it:

$$\begin{aligned} \lambda r.\ &\mathbf{if}\ r.\mathsf{shell} = \texttt{"/usr/bin/scsh"} \\ &\mathbf{then}\ \mathit{notify}\ r.\mathsf{user} \\ &\mathbf{else}\ \langle\rangle. \end{aligned}$$

Clearly r is a record that has both of the mentioned fields, and it may have others as well, so it will have a type scheme like

$$\langle \mathsf{user} : string; \mathsf{shell} : string; \rho \rangle.$$

Row types may also support extension and concatenation. For example, given a record r that has a field $\mathsf{user} : string$, we may wish to extend it by a field with the remaining quota of the user's account, leaving other fields unchanged:

$$\lambda r.\ r\ \mathbf{with}\ \mathsf{quota} = \mathit{userquota}\ r.\mathsf{user}.$$

Whatever the argument type, the result type should be the same, except that now it has the field $\mathsf{quota} : int$ as well. What if it had a quota field already? Depending on the particular system, we may need to add a side constraint that eliminates or otherwise regulates this possibility, so we give the function a type like

$$\langle \mathsf{user} : string; \rho \rangle \rightarrow \langle \mathsf{user} : string; \mathsf{quota} : int; \rho \rangle \quad [\mathsf{quota}\#\rho],$$

where $\mathsf{quota}\#\rho$ is the constraint that ρ does not include the label quota.

4.2 Row Types in OCaml

When designing Shcaml, we intended to use OCaml's row types to implement structural subtyping and extension of line records. We were in for an unpleasant surprise.

OCaml's row types are restricted by a simple rule: if an extension variable would appear in two row types, then both rows are the same (Rémy and Vouillon 1998). Equivalently, extension variables in OCaml are anonymous, since there is no need to name them in order to relate two different rows; thus, all extension variables are written as `..` in OCaml. This choice effectively disallows extension and concatenation, because it cannot express the notion that row types share some but not all of their fields.

Rémy and Vouillon explain this design decision in their implementation of Objective ML, of which modern OCaml is a direct descendant. Because their language does not provide primitive operations for extension and concatenation, they write,

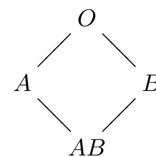
> [t]hese types can thus be ruled out without seriously restricting the language. Moreover, all programs keep the same principal types. This restriction was implemented to avoid explaining sorts to the user. It also makes the syntax for types somewhat clearer, as row variables can then always be replaced by ellipsis. Furthermore, sharing can always be described with aliasing. For instance, $\langle m : \tau; \rho \rangle \rightarrow \langle m : \tau; \rho \rangle$ is written $(\langle m : \tau;\ .. \rangle\ \texttt{as}\ \alpha) \rightarrow \alpha$.

The consequences for Shcaml are dire: OCaml's row types cannot express extensible line records.

4.3 Faking It

While we cannot have real record extension in Shcaml, it is possible to fake it, provided we assume a finite, closed world of field labels. Our technique is essentially as described in Rémy (1993). We use a phantom type encoding reminiscent of Fluet and Pucella (2006) to embed Rémy's type system in OCaml.

The encoding uses a phantom type parameter of k variables to simulate a complete subtype lattice of 2^k elements. For example, suppose we wish to express a subtyping lattice with four elements:



We require a phantom type with two variables, which we might construe as "presence of A" and "presence of B," analogous to Rémy's distinguished type constructors *pre* and *abs*. In Shcaml, we use an OCaml object type as the phantom parameter, since that makes it easy to see which field is which[4].

Fluet and Pucella's treatment in Standard ML also requires two abstract types, $\mathit{present}$ and absent. Types in the lattice are represented differently depending on whether they occur in negative

[4] The choice to use OCaml object types as a phantom parameter was motivated by the particular type errors they induce. Type errors in Shcaml can at times be rather large, but they are usually comprehensible.

<table>
<tr><th>Fitting combinator</th><th>Usage</th><th>Shell equivalent</th></tr>
<tr><td><code>(-|)   : ('i -> 'm) t -> ('m -> 'o) t -> ('i -> 'o) t</code></td><td>c_1 <code>-|</code> c_2</td><td>c_1 <code>|</code> c_2</td></tr>
<tr><td><code>(/</) : (text -> 'o) t -> dup_in_spec -> (text -> 'o) t</code></td><td>c <code>/</[</code>d_1<code>%<&</code>d_2<code>]</code></td><td>c d_1<code><&</code>d_2</td></tr>
<tr><td><code>(/>/) : ('i -> 'o elem) t -> dup_out_spec -> ('i -> 'o elem) t</code></td><td>c <code>/>/[</code>d_1<code>%>&</code>d_2<code>]</code></td><td>c d_1<code>>&</code>d_2</td></tr>
<tr><td><code>(^>>=) : ('i -> 'o) t -> (Proc.status -> ('i -> 'o) t) -> ('i -> 'o) t</code></td><td>c <code>^>>= fun res -></code> ...</td><td>c<code>; res=$?</code> ...</td></tr>
<tr><td><code>(^>>) : ('i -> 'o) t -> ('i -> 'o) t -> ('i -> 'o) t</code></td><td>c_1 <code>^>></code> c_2</td><td>c_1<code>;</code> c_2</td></tr>
<tr><td><code>(&&^) : ('i -> 'o) t -> ('i -> 'o) t -> ('i -> 'o) t</code></td><td>c_1 <code>&&^</code> c_2</td><td>c_1 <code>&&</code> c_2</td></tr>
<tr><td><code>(||^) : ('i -> 'o) t -> ('i -> 'o) t -> ('i -> 'o) t</code></td><td>c_1 <code>||^</code> c_2</td><td>c_1 <code>||</code> c_2</td></tr>
<tr><td><code>(~>>) : ('i -> 'o) t list -> ('i -> 'o) t</code></td><td><code>~>>[</code>c_1<code>;</code> c_2<code>;</code> c_3<code>; ...]</code></td><td>c_1<code>;</code> c_2<code>;</code> c_3...</td></tr>
<tr><td><code>(~&&) : ('i -> 'o) t list -> ('i -> 'o) t</code></td><td><code>~&&[</code>c_1<code>;</code> c_2<code>;</code> c_3<code>; ...]</code></td><td>c_1 <code>&&</code> c_2 <code>&&</code> c_3...</td></tr>
<tr><td><code>(~||) : ('i -> 'o) t list -> ('i -> 'o) t</code></td><td><code>~||[</code>c_1<code>;</code> c_2<code>;</code> c_3<code>; ...]</code></td><td>c_1 <code>||</code> c_2 <code>||</code> c_3...</td></tr>
<tr><td><code>(^&=) : (text -> 'b elem) t -> (Proc.t -> ('i -> 'o) t) -> ('i -> 'o) t</code></td><td>c <code>^&= fun pid -></code> ...</td><td>c <code>& pid=$!</code> ...</td></tr>
<tr><td><code>(^&)   : (text -> 'b elem) t -> ('i -> 'o) t   -> ('i -> 'o) t</code></td><td>c_1 <code>^&</code> c_2</td><td>c_1 <code>&</code> c_2</td></tr>
</table>

Table 2. Some of Shcaml's fitting combinators

or positive position. In positive position, a type variable denotes presence and the type *absent* to denote absence, whereas in negative position, *present* denotes presence and a type variable denotes absence. For example, to represent the function type $A \to B$, we use the type

$$\langle \mathsf{A} : \mathit{present}; \mathsf{B} : \beta_1 \rangle\, t \to \langle \mathsf{A} : \mathit{absent}; \mathsf{B} : \beta_2 \rangle\, t.$$

We can read this as saying that the argument *must* have A, and may or may not have B, and that the result *does not* have A, and may be treated as if it has B or not.

In OCaml, however, we can make *present* a subtype of *absent* directly, which means that the positive versus negative distinction is no longer necessary, and we represent the type $A \to B$ as

```
<a : present; b : absent> t
        -> <a : absent; b : present> t
```

Shcaml's line records assume a closed world of labels, and assume (for the most part) that each label may be associated with only one type. It also assumes that *every* line record carries a string representing its "main" value, which is the component of the line record that is sent to an external process, if necessary. Shcaml implements a domain-specific language in which to specify the world of labels, from which it generates a line record implementation at compile time. We might, for example, declare a world of three labels:

```
val a : int
val b : bool
val c : string
```

This generates selectors and functional updaters for each field:

```
a : <a : present; b : 'b; c : 'c> line -> int
b : <a : 'a; b : present; c : 'c> line -> bool
c : <a : 'a; b : 'b; c : present> line -> string

set_a : int -> <a : 'a; b : 'b; c : 'c> line
          -> <a : present; b : 'b; c : 'c> line
set_b : bool -> <a : 'a; b : 'b; c : 'c> line
          -> <a : 'a; b : present; c : 'c> line
set_c : string -> <a : 'a; b : 'b; c : 'c> line
          -> <a : 'a; b : 'b; c : present> line
```

Because those types are somewhat unwieldy, we implemented a syntax extension that simulates extension variables. Using the extension, we may write the types of `a` and `set_a` as

```
a     : <a : present; ..> line -> int
set_a : int -> <a : 'a; .. as 'r>
          -> <a : present; .. as 'r>
```

Shcaml's line record DSL also supports nested records, which may include mandatory fields. For example, the basis world includes a nested line record `Passwd`, which contains fields from the `/etc/passwd` file. While a `Passwd` subrecord may be present or absent, if it is present, then all its fields are present as well. Shcaml comes with a large initial predefined world, which in addition to `/etc/passwd` records, includes key-value pairs, arbitrary tabular data (with optional field names), `/etc/group` records, the `stat` structure for file metadata, file mode bits, process metadata from `ps`, `/etc/fstab` records, and `mailcap` entries. Some of these line records are not read from text files but built from the result particular system calls; in general, line records may be constructed by arbitrary computations.

5. Fittings

Given a value `is : int shtream -> string shtream`, that is, a process that consumes values of type `int` to produce values of type `string`, and given another process `sb : string shtream -> bool shtream` that consumes `string`s to produce `bool`s, we may compose them into a process that consumes `int`s and produces `bool`s:

```
let ib (src : int shtream) : bool shtream = sb (is src)
```

We may then apply `ib` to an `int shtream` to construct a `bool shtream`.

This simple function composition story is sufficient to express many typed pipelines, but there are other things we cannot easily do with processes composed in this manner that are simple in shell:

- Shell commands may have their input and output redirected.
- Shell commands may be sequenced, such that a sequence of processes share the same input and output at their spot in the pipeline.
- Shell commands may be executed conditionally based on the result code of other processes.

Beyond these semantic considerations, we believe it is syntactically advantageous to write long pipelines in diagrammatic (left-to-right) rather than standard composition order.

Fittings, Shcaml's process notation, are intended to solve these problems. A fitting of type `('a -> 'b) Fitting.t` generally represents an `'a shtream -> 'b shtream` function. Fittings are composed in diagrammatic order with the fitting pipe operator `(-|)`, and provide combinators for redirection, sequencing, and conditionals. Fittings are first class objects, and a variety of functions may be used to actually run them in varying contexts.

While fittings are notionally similar to shtream-to-shtream functions, fitting composition is not merely function composition. Rather, it constructs a data structure that allows particular combinations to be specialized when fittings are run. For example, the fitting `to_file f` writes its input out to the file `f`. If this fitting is composed on the right with a fitting that produces an internal

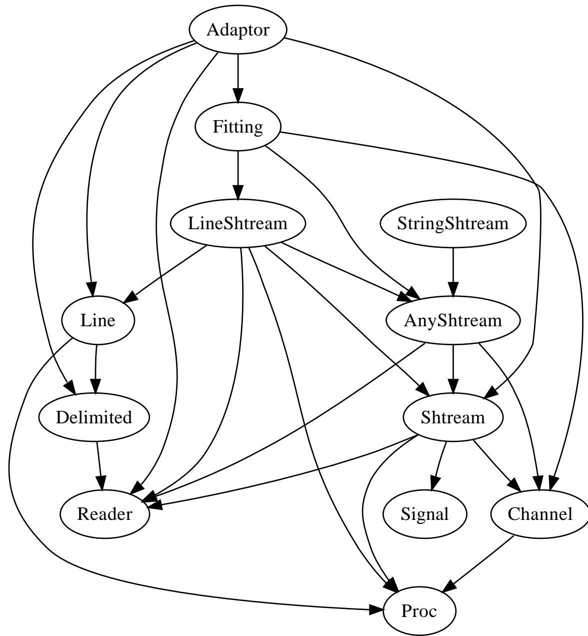


Figure 4. Module dependency graph for Shcaml

shtream, then it opens the file, reads each element from its input shtream, and writes each element to the file. If its input comes from an external process, however, `to_file` opens the file *before* the process on its left is started, so that the process's standard output may be redirected to the file directly, avoiding the need for OCaml to shuttle the data between the external process and the file. This may seem like a minor optimization, but it actually has important semantic consequences. Suppose, for example, that one wants to run a program interactively from within a fitting, even if the current redirections do not point to the terminal:

```
from_file "/dev/tty"
  -| command "ssh example.com" />/ [ 2%>&1 ]
  -| to_file "/dev/tty"
```

If the command's standard output is a not a terminal, its output is buffered for efficiency, which means that the user may see responses not when they are produced but when the buffer eventually is flushed. Thus, it is important that `to_file` did not connect the command's standard output directly to `/dev/tty`, rather than shuttle the data through a pipe from `ssh` and then to `/dev/tty`.

Fitting combinators. In addition to the pipe operator `(-|)`, Shcaml provides fitting operators for redirection, conditionals, sequencing, and starting background processes (Table 2).

The redirection operators `(/>/)` and `(/</)` add input and output redirections, respectively, to a fitting. Each takes a fitting and a list of redirection specifiers such as `[ 2 %>& 1; 1 %> "out" ]`, which sends standard output to the file `out` and redirects errors to the standard output.

Conditionals are derived from the conditional operator `(~>>=)`,[5] which takes a fitting to run and a continuation that receives the first fittings result code and then returns the next fitting to run. Both the standard shell conditional operators, `(&&^)` and `(||^)`, and the sequencing operator `(^>>)` are easily defined in terms of `(~>>=)`:

```
let (&&^) a b = a ^>>= function
  | Proc.WEXITED 0 -> b
  | n              -> yield n
let (||^) a b = a ^>>= function
  | Proc.WEXITED 0 as n -> yield n
  | _                   -> b
let (^>>) a b = a ^>>= fun _ -> b
```

List versions of the conditional and sequencing operators are available as well.

The background operator `(^&=)` is used in continuation-passing style just as `(^>>=)` is, but it passes its continuation a process ID rather than an exit code. Operator `(^&)` combines two fittings, running its left argument in the background but not making its PID available.

Fitting runners. Shcaml provides several functions for running fittings:

`run`	evaluates a fitting connected to the standard input and output, waits for it, and returns its exit status
`run_bg`	evaluates a fitting connected to the standard input and output, returning its process ID
`run_source`	evaluates a fitting connected to the standard input and returns a shtream of its output
`run_sink`	returns a `coshtream` object that allows writing to the fitting's input
`run_shtream`	renders a fitting as a shtream-to-shtream function
`run_in`	run a fitting, returning its input as an `in_channel`
`run_out`	run a fitting, returning its output as an `out_channel`
`run_backquote`	returns the output of a fitting as a string

5.1 Readers, Splitters, and Adaptors

A principal design goal for Shcaml is to change data from untyped, unstructured strings to useful structured data as early and easily as possible. Two Unix programs will happily communicate directly through a pipe, and two internal stream transducers will pass OCaml values with no need of marshalling. But when the output from an external program flows to the input of an internal transducer, we require some mechanism to turn bytes into meaningful data that is easy to work with in OCaml. We factor this work into two steps, first identifying the boundaries of records, and second splitting records into fields.

The first task, extracting records from an input channel, is performed by *readers*. Intuitively, readers allow the programmer to treat any output format as if it were "line-oriented." For many data formats, records are delimited by newlines, but Shcaml readers are not constrained in this way—they are merely passed an input channel and must produce a `raw_line` record:

```
type raw_line = { content : string;
                  before  : string;
                  after   : string; }
```

This definition reveals readers also to have a more subtle responsibility in Shcaml than just reading files. Because the programmer may compose fittings arbitrarily (up to well-typedness), it is essen-

[5] The strong resemblance many of our operator symbols bear to line noise is not intended to remind users of the lexical syntax of another popular scripting language, but is due to OCaml's operator precedence and fixity rules.

tial that text read and parsed from a Unix program retain sufficient information as to reproduce it exactly. Readers keep track of not only the portion of each line that contains the data, but also the text they trim or ignore.

Once data has been split into `raw_line` records by a reader, it must be parsed into meaningful OCaml data, such as a line record. A splitter takes a `raw_line` and attempts to parse it into a line record of the appropriate type—for example, the password splitter takes a `raw_line` and, if successful, returns a line record containing the parsed password fields.

An *adaptor* combines a fitting for splitting a particular format along with a "reader hint" for the reader appropriate to that format. For example, the password adaptor has type

```
(< passwd: absent;  .. as 'r > line ->
 < passwd: present; .. as 'r > line) fitting
```

which is a fitting that consumes line records with no password data (and perhaps other fields) and produces line records with the password data parsed from their "main" string (and the same other fields).

The password adaptor has another effect as well: it sends a hint to the upstream fitting that the source should use a reader appropriate for password files. If the user has specified another reader explicitly when reading from the file, this will not override it and the user's specified reader will be used. If the user does not explicitly specify a reader, then an adaptor causes a reasonable default reader for its file format to be used.

6. Processes and Channels

Shcaml's higher-level abstractions (such as fittings) are based on mid-level abstractions around Unix processes and file handles. Shcaml's `Proc` module, for example, provides easier management of subprocesses.

In Unix, new processes are created by the *fork*() system call, which returns the process ID of the new child process to the parent. After a successful call to *fork*(), the parent process may call *wait*() or one of its variants (*e.g. waitpid*()), which waits for a child (or the specified child) to terminate and returns its exit status. There are several rules that govern how this works:

- By default, each child's process metadata is retained by the system, pending the parent's call to *wait*(); thus, if the parent does not wait, the process table fills with garbage.
- The parent may elect to have all children removed automatically from the process table, thereby losing access to their exit statuses.
- The parent may wait on each child no more than once, since waiting causes the child to be removed from the process table.

When a user starts programs from a shell, the shell takes care of all this, but when a programmer starts child processes by calling *fork*() directly, it may be difficult to get right.

Shcaml provides a better interface, which is based on Scsh (Shivers 1994). `Proc.fork` returns a `Proc.t` value, which can be used to wait for the child process to exit and return its status, or to retrieve its exit status only if the child has exited, in a non-blocking manner. Because Shcaml waits on every child and stores its exit status in a weak hash table, there is no *need* for the parent to wait—if it drops all references to the `Proc.t` value, then Shcaml will remove the child from its own process table. Conversely, so long as the `Proc.t` exists, the status may be retrieved as many times as necessary from Shcaml's process table.

The `Channel` module enriches OCaml's standard I/O to make it both more flexible and easier to manage. In OCaml, `in_channels` support only reading and `out_channels` support only writing. Because they have different types, attempting to write to an `in_channel` is a static error. In the Unix world, however, a process accesses files through *file descriptors*, which are merely integer indices into a table of open files. Unix file descriptors may be open for reading, writing, or both, and attempting to write to a read-only descriptor is a dynamic error. Thus, we actually have two mismatches:

- File descriptors fail to catch some erroneous uses at compile time that OCaml's channels prohibit via the type system.
- OCaml's channels are unidirectional, which means they do not directly support the bidirectional aspect of file descriptors.

Furthermore, when writing systems programs it is often necessary to work directly with file descriptors rather than channel abstractions, because some features (such as redirection) are specified in terms of file descriptors. Sometimes, when interacting with the shell, for example, even abstract file descriptors are insufficient, and knowledge of the actual integer values is necessary.

Shcaml's `Channel` module solves the mismatch problem and supports the low-level use cases. We take advantage of OCaml's open variants so that it is easy to express various combinations of I/O possiblities:

```
type gen_in_channel   = [ `InChannel  of in_channel
                        | `InDescr    of descr
                        | `InFd       of int ]

type gen_out_channel  = [ `OutChannel of out_channel
                        | `OutDescr   of descr
                        | `OutFd      of int ]

type gen_channel      = [ gen_in_channel
                        | gen_out_channel ]
```

Many of Shcaml's channel operations operate on the precise type appropriate for what they do. Using these types, it is possible to specify that an operation requires a file descriptor with a particular mode, or to allow an operation to take either an input or output channel.

Shcaml also makes starting external processes and communicating with them over pipes both easier and more flexible than C or OCaml's `Unix` module. Writing in either C or OCaml, there are essentially two ways to start an external process.

If we want a certain level of control, for example, if we want to know the PID of the child in order to signal it, we may use the low-level system call interface. This requires explicit creation of a pipe, forking, redirection, and execution of the new program; then, when the parent is done talking to the subprocess, it needs to close the pipe and wait for it to exit. We demonstrate this (without the proper error checking) in Figure 5(a). To acheive the same level of control in OCaml, the code we write is largely the same as in C (Figure 5(b)). Shcaml, on the other hand, provides detailed control with a high-level library call (Figure 5(c)). Shcaml's process and pipe creation functions also accept optional arguments for setting up addition pipes and redirections in the child process, modifying the search path, or running arbitrary OCaml code in the subprocess.

If we value convenience over control, both C and OCaml offer a library call that takes care of the details for us (figures 5(d) and 5(e)). Unfortunately, in this case we cannot discover the PID of the child process. More disconcertingly, while C's `popen` (resp., OCaml's `open_process_in`) appears to return a normal `FILE*` (`in_channel`), it cannot be closed with the usual `fclose` (`close_in`). To close it properly we must use `pclose` (`close_process_in`). Shcaml provides a similar high-level API (Figure 5(f)), but the resulting `in_channel` may be closed with `close_in`, which means that there is no need to keep track of which channels are connected to files and which to processes.

```
int fd[2]; pipe(fd);
int pid = fork();

if (pid) { /* parent */
  close(fd[1]);
  /* ··· */
  close(fd[0]);
  waitpid(pid, &status, 0);
} else {   /* child  */
  close(fd[0]);
  dup2(fd[1], 1);
  execlp("ls", "ls", "-l", NULL);
  exit(1);
}
```

(a) Low-level C

```
let (fd0,fd1) = pipe () in
let pid       = fork () in

if pid <> 0 then (* parent *)
  (close fd1;
   (* ··· *);
   close fd0;
   waitpid [] pid)
else             (* child  *)
  (close fd0;
   dup2 fd1 stdout;
   execvp "ls" [| "ls"; "-l" |];
   exit 1)
```

(b) Low-level OCaml

```
let procref = ref None in
let ic = open_program_in
           ~procref
           "ls" ["-l"] in
(* ··· *)
close_in ic
```

(c) Shcaml

```
FILE *ifd = popen("ls -l", "r");
/* ··· */
pclose(ifd);
```

(d) Using the C library

```
let ic = open_process_in "ls -l" in
(* ··· *)
close_process_in ic
```

(e) Using the OCaml library

```
let ic = open_command_in "ls -l"
(* ··· *)
close_in ic
```

(f) Shcaml

Figure 5. Piping from external processes: C vs. OCaml

Furthermore, Shcaml's high-level interface *is the same* as its high-control interface, providing detailed control via optional arguments.

7. Some More Examples

In this section, we demonstrate two example scripts written in Shcaml, in order to give a better sense of the library.

Process grep. Some Unix systems come with the utility pgrep, which examines the process table and returns the process IDs of running commands whose name matches its argument. While developing Shcaml, we noticed that the Unix systems that we were using did not have pgrep. A few lines of code solved that problem. The ps function above is a convenience function provided with Shcaml that will generate a shtream of lines containing data parsed from the Unix ps program, from which we then filter with grep and project out the correct field:

```
let pgrep pat = run begin
  ps () -|
  grep (Reader.starts_with pat ∘ Line.Ps.command) -|
  cut (string_of_int ∘ Line.Ps.pid)
end
```

Racing processes. The following code runs a collection of processes concurrently, and when one of the processes finishes, it kills the others. This provides funtionality akin to McCarthy's ambiguous function operator, *amb* (1963), but for Unix shell processes.

The program is factored into two functions. The function bg_list takes a list of commands to run in parallel and a continuation to which it passes the list of their process IDs (as Proc.ts):

```
let rec bg_list commands kont =
  match commands with
  | []      -> kont []
  | x :: xs -> command x ^&= fun proc ->
               bg_list xs ^$ fun procs ->
               kont (proc :: procs)
```

The function race takes a list of commands, which it passes to bg_list to start them in the background. It uses wait_any to wait until one of them exits (we do not care about the exit status), sends a signal to kill all of them, and then exits successfully:

```
let race commands = run begin
  bg_list commands ^$ fun procs ->
  ignore ^$ Proc.wait_any procs;
  List.iter (Proc.kill ~raise:false 9) procs;
  yield (Proc.WEXITED 0)
end
```

8. An Informal Benchmark

> The programmer grew distressed. "But through the C language we experience the enlightenment of the Patriarch Ritchie! We become as one with the operating system and the machine, reaping matchless performance!"
>
> Rootless Root: The Unix Koans of Master Foo

We designed a simple benchmark for Shcaml to get a feel for how it compares to other systems in performance and ease of use. For each benchmark, we use the average of ten trials, run in single-user mode on an Apple MacBook with 2 GHz Intel Core 2 Duo processor and 2 GB of memory. The benchmark is to read a file containing 100 copies of our /usr/dict/words (23,493,600 lines), retain only lines matching a particular regular expression, drop lines matching a different regular expression, and then count the number of lines remaining.

We chose this benchmark because it requires processing a nontrivial amount of data, and because it is readily implemented as a pipeline which must be passed through several pipes.

The baseline implementation (sh-pipe) is in shell:

```
cat words          |
  grep '.aaa*.'    |
  grep -v '.ooo*.' |
  wc -l
```

This script works by delegating to four subprocesses connected by pipes. First, cat reads the file and writes it to the pipe that the first grep above reads from. That grep filters out all words which do not contain a substring matching the regular expression .aaa*. (a single character, two or more as, followed by another character). This filtered data is passed to another instance of grep, which removes lines that match its pattern, .ooo*.. Finally, the data is piped to wc -l, which counts the number of lines and prints the result.

```
run begin
  command "cat words" -|
  command "grep '.aaa*.'" -|
  command "grep -v '.ooo*.'" -|
  command "wc -l"
end
```

(**a**) shcaml-pipe

```
(run
  (| (cat words)
     (grep .aaa*.)
     (grep -v .ooo*.)
     (wc -l)))
```

(**b**) scsh-pipe

Figure 6. Pipeline-based benchmark implementations

```
my $count = 0;

open STDIN, "words";

while (<>) {
  if (/.aaa*./o and not /.ooo*./o) {
    ++$count;
  }
}

print "   $count\n";
```

(**a**) perl-int

```
let aaastar = Str.regexp ".*.aaa*." in
let ooostar = Str.regexp ".*.ooo*." in

let count = ref 0 in

Shtream.iter {| incr count |} (run_source begin
  from_file "words" -|
  grep (fun str ->
          Str.string_match aaastar str 0) -|
  grep (fun str ->
          not (Str.string_match ooostar str 0))
end);

print_endline ("   " ^ string_of_int (!count))
```

(**b**) shcaml-int

```
stringbuf buf;
regex aaastar(".aaa*."), ooostar(".ooo*.");
int count(0);
ifstream f("words");

while (f.good()) {
    buf.str("");
    f.get(buf).get();
    string s = buf.str();
    if (aaastar.match(s) && !ooostar.match(s))
      ++count;
}

cout << "   " << count << endl;
```

(**c**) c++-int

```
(define .aaa*. (rx (: any "aa" (* "a") any)))
(define .ooo*. (rx (: any "oo" (* "o") any)))

(call-with-input-file "words"
  (lambda (fd)
    (port-fold
      fd
      read-line
      (lambda (line count)
        (if (and (regexp-search? .aaa*. line)
                 (not (regexp-search .ooo*. line)))
            (+ 1 count)
            count))
      0)))
```

(**d**) scsh-int

Figure 7. Single-process benchmark implementations

```
run begin
  command "cat words"        -|
  sed id                     -|
  command "grep '.aaa*.'"    -|
  sed id                     -|
  command "grep -v '.ooo*.'" -|
  sed id                     -|
  command "wc -l"
end
```

(**a**) shcaml-mix

```
(run
  (| (cat words)
     (begin (copy-lines))
     (grep .aaa*.)
     (begin (copy-lines))
     (grep -v .ooo*.)
     (begin (copy-lines))
     (wc -l)))
```

(**b**) scsh-mix

Figure 8. Benchmark implementations with a mixed pipeline

implementation	real (s)	user (s)	sys (s)
shcaml-pipe	0.82	0.75	0.40
sh-pipe	0.82	0.75	0.38
scsh-pipe	0.85	0.77	0.40
perl-int	9.52	9.18	0.33
shcaml-int	17.61	17.11	0.49
c++-int	24.14	23.25	0.88
scsh-int	938.41	902.11	36.28
shcaml-mix	122.85	79.83	146.21
scsh-mix	1026.71	1016.07	20.67

Table 3. Benchmark results

We implemented this benchmark using essentially three strategies:

- In Shcaml and Scsh, we translated the shell script directly; each of these versions, like the shell version, starts the four subprocesses and then waits (shcaml-pipe and scsh-pipe, Figure 6). Consequently, they perform almost the same as the shell version.
- In Perl, Shcaml, Scsh, and C++, we implemented the program directly, without delegating to subprograms (perl-int, shcaml-int, scsh-int, and c++-int, Figure 7). Each of these programs reads lines in a loop, matches using a regular expression library, and counts. These programs perform significantly worse than the pipeline-based implementations. Almost all of the overhead, it seems, is due to these languages' slow implementations of file I/O—these programs spend significant time just reading lines from the input file. In this category, Perl was the clear winner, with Shcaml taking twice as long as Perl, C++ taking almost three times as long, and Scsh the slowest.
- In Shcaml and Scsh, we implemented the benchmark as a pipeline intermixing external unix processes and data-shuttling code written in the host language (shcaml-mix and scsh-mix, Figure 8). These programs performed extremely poorly compared to the other implementations. In fact, Scsh crashed in seven out of then 10 trials. Much of the time, again, appears to be spent in the host languages' I/O systems.

Table 3 shows the mean user, system and wall-clock time over 10 runs of each implementation of the benchmark (except for scsh-mix, which only completed three times). From these results, we can see that the pipeline-based Shcaml implementation is competitive with the original shell version and the comparable Scsh implementation. This is reassuring, as it suggests that Shcaml introduces virtually no overhead when simply acting as a shell. Certainly, the results we observed for several of the benchmark runs would not have been so dramatic if the various language implementations we tested had faster I/O routines. These timings suggest that I/O inefficiency is a greater performance bottleneck than any inefficiencies introduced by Shcaml.

We consider these results evidence that the ability to integrate external Unix processes into larger programs simply and seamlessly is a useful and effective tool. Not only does Shcaml make it easy to take advantage of the functionality of extant programs on the system, it also helps programmers write more efficient code than they might have otherwise. Traditionally, Unix programs have been designed to be small utilities that do a single thing very well. If an application can take advantage of this software in a convenient way, it may not only be easier to write, but indeed perform competitively or even flat-out dominate ad-hoc code written to achieve the same task.

9. Conclusion

> "And who better understands the Unix-nature?" Master Foo asked. "Is it he who writes the ten thousand lines, or he who, perceiving the emptiness of the task, gains merit by not coding?"
>
> Rootless Root: The Unix Koans of Master Foo

Shcaml is a rich library[6] that unifies OCaml and shell programming. It provides versatile and powerful functionality at several levels of abstraction, which may be used individually or in concert to make it easier for programmers to control Unix processes. Mid-level abstractions found in Shcaml's `Channel` and `Proc` modules provide a uniform interface to the primitive computational constructs provided by the operating system. Data shtreams allow the programmer to conveniently and efficiently construct pipelines that freely intermix OCaml code with external Unix processes. Complex pipelines may be created and manipulated through fittings, which provide a combinator library very close to the shell's command language. Fittings are designed to work with special line record values whose types express precise information about the data flowing through pipelines.

In contrast to the conventional library of operating system calls, Shcaml provides an expressive and convenient means to take advantage of Unix and shell programming idioms inside a typed, functional programming language. However, it does not currently enable the converse functionality: A terse, pragmatic, and effective syntax for calling into OCaml from an interactive command shell. In the future, we would like to develop a syntax for an interactive command shell that could be translated into Shcaml.

Shcaml is not a small program. It comprises roughly 8000 lines of commented Objective Caml. Some aspects of the functionality that we implemented in order to make Shcaml viable were concrete instances of language technologies that have existed for over a decade. For example, the implementation of line records, which weighs in at a hefty 1400 lines, would be rendered obsolete (and in fact quite inferior) if OCaml's row types supported extension.

People talk a lot these days about "scripting languages." We propose that Shcaml points toward a new sweet spot in scripting language design. Writing short shell scripts in OCaml is now as smooth as writing large applications, and integrating the two is easy and intuitive. OCaml programs scale to large applications with thousands of lines; with Shcaml, they also scale down to just the right few. We have shown that there is no conflict between easy access to system facilities and advanced type and module features; they can coexist or even complement one another.

Acknowledgments

Thank you to Dan Brown, Jed Davis, Riccardo Pucella, Sam Tobin-Hochstadt, and Aaron Turon for their thoughtful comments, advice, and encouragement on many drafts of this paper. Many thanks also to Jane Street Capital for initially sponsoring this project as part of their OCaml Summer of Code program.

References

Standard for information technology – portable operating system interface (POSIX). shell and utilities. *IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard. Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004. Shell and Utilities*, 2004.

S. R. Bourne. An introduction to the UNIX shell. *Bell System Technical Journal*, 57(6):1971–1990, 1978.

[6] Source code and documentation may be found at `http://www.ccs.neu.edu/home/tov/shcaml/`.

M. Fluet and R. Pucella. Phantom types and subtyping. *Journal of Functional Programming*, 16(2):751–791, 2006.

J. McCarthy. A basis for a mathematical theory of computation. *Computer Programming and Formal Systems*, pages 33–70, 1963.

A. J. Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, 17(9): 7–13, 1982.

D. Rémy. Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4:27–50, 1998.

Rootless Root: The Unix Koans of Master Foo. In E. S. Raymond. *The Art of Unix Programming.* Addison-Wesley, 2004.

O. Shivers. A Scheme shell. Technical report, Massachusetts Institute of Technology, 1994.

J. Snover. Monad manifesto. Technical report, Microsoft, August 2002.

B. Verlyck. *Cash, the Caml Shell*. INRIA, 2002. URL `http://pauillac.inria.fr/cash/`.

C. G. Walters. Hotwire shell, 2007. URL `http://hotwire-shell.org/`.

M. Wand. Complete type inference for simple objects. In *Proc. 2nd Annual IEEE Symposium on Logic in Computer Science (LICS'87)*, pages 37–44, 1987.