Stream Input/Output

Luca Cardelli

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

This is a proposal for sequential Input/Output in Standard ML. My ideas on I/O have been deeply influenced by the Unix operating system, however I do not consider this as an uncritical suggestion of doing I/O "like in Unix" (e.g. Common Lisp turns out to be particularly close to this proposal). I have been under the influence of other operating systems for longer periods of time, and I have often tried to figure out nice sets of I/O primitives, but nothing came out of it. If some ideas have come out of Unix, I like to think it is because they are inherently simple and powerful.

I think I now understand that I/O issues are often obfuscated by convoluted abstractions and misplaced efficiency considerations. Even when the right primitives are available, they are hidden under layers of software. In a few words, most of the time they have got it wrong.

Programming languages fall into two categories with respect to I/O. System programming languages often directly call the operating system through bizarre parameter lists, offering a jungle of 'primitives' with no mediating abstraction. On the other hand, I/O systems in algorithmic languages are often overrestricted and oversimplified toys (e.g. Pascal), meant as the intersection of all possible I/O systems. Very few languages have files as first class objects, which is the single most important requirement for any non-trivial use of I/O.

It is time to include powerful I/O systems *within* programming languages, particularly because Unix-like operating systems, which can easily support them, are becoming universal standards, and most modern operating systems have the necessary primitives. Crude I/O systems are particularly crippling on personal computers, where many common I/O limitations simply do not apply.

Files and processes

Streams are communication windows on the external world of *files* and *processes*. The basic design goal of this proposal is the following:

> External files and processes should offer a uniform interface.

The above property entails great simplicity and uniformity in operating systems, and has particularly been explored in Unix. Unix is totally based on the *file* paradigm: everything is a file; even a terminal process "is" a pair of files because it is identified with its I/O streams. Similarly, everything could be based on the *process* paradigm. There a file "is" the process which reads and writes it.

The important point is the consistent view of the outside world. If the file and process interfaces

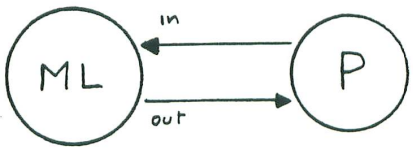
---

Unix is a trade mark of Bell Laboratories

are uniform, then programs can be written without embedding knowledge of *who* or *what* they will be talking to. This makes them very flexible and usable in different and unpredictable situations.

Files can be *read* and *written*; temporary files are often alternatively read and written. Processes can accept *questions* and provide *answers*: interesting processes require a dialogue. This suggests (even if it does not demand) that communication streams should be *bipolar*, admitting both input and output actions(1). Bipolar temporary files can be implemented very efficiently by keeping them in memory most of the time. Bipolar channels to other processes can be set up with cleaner protocols than pairs of unipolar channels. A typical bipolar file is one used as an interface between two passes of a compiler. A typical bipolar channel is *terminal*, for terminal I/O (as opposed to unipolar input and output streams).

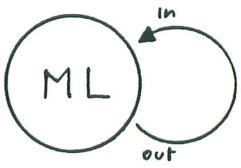
A bipolar stream connected to a process looks like this:



Such a stream is also called a *channel*, or an *external* stream. P may be another ML system.

Any output operation should always be immediately successful. An input operation may have to wait for P to produce some input: in this case the input operation should *hang*. The alternative to hanging is failing on empty stream: this requires *polling* of the input (to discover when data is ready), which is bad style and it is either wasteful (for frequent polling), or it degrades the interaction (for unfrequent polling, e.g. when P is the terminal). To detect the possibility of hanging, it is then necessary to be able to test for empty stream.

A bipolar stream connected to a file is obtained by letting P be the identity process (for an initially empty file) or a process which executes a single initial output, and then becomes the identity (for a file with some initial contents):



Such a stream is also called a *file* or an *internal* stream (as no external process is actually involved). The above discussion implies that input operations should hang on empty files, even if in this situation the wait would never end(2).

Some stream can be unipolar. I/O operations fail on these stream streams if they have the wrong polarity.

---

(1) Bipolar files are called *pipes* in Unix, and *bidirectional streams* in Common Lisp. This proposal does not in any way assume the existence of pipes, and even under Unix will not be implemented by pipes. Bipolar files are simply pairs of file descriptors pointing to the same file.

(2) The "break" key can be used to stop infinite waits, producing an "interrupt" exception.

External processes

Why should we bother including arbitrary external processes in an I/O system? Aren't files enough?

So far I have tried to show how natural and compelling is the extension of file primitives to processes. Another important observation(3) is that process channels can put a stop to the dozens of meaningless extensions that people will inevitably end up hacking in any ML system. For example somebody may require a "date" primitive, or want to read his mail from ML.

The *dirty* alternatives are: (a) put "date", "mail", etc. primitives in the ML compiler; (b) supply a general "external function call" feature to call the date or mail routines, or any routine, endangering the ML system and undermining its type security; (c) introduce an operating system escape mode.

The *clean* alternative is to dynamically create channels to the "date" and "mail" *processes*: this way all the possible interaction goes through character channels which guarantee the isolation and protection of ML, while allowing two-way communication.

Will external processes change the ML programming style? I do not think so: it will just make possible things which are now impossible or awkward. The "whole world" outside ML will be available from the system by a clean, powerful and secure set of primitives, even if at the moment we do not know what this external world will look like.

Can I implement it?

This I/O system is not implementable in many high level languages (e.g. Pascal), essentially whenever files are not objects in the language. However most operating systems provide the necessary primitives. Basically, it is necessary to be able to open, close, read and write simple character files, and to position at random locations in a file. It must also be possible to perform unbuffered I/O to processes and devices, and to obtain and store file descriptors. Multiple file descriptors for the same file are very convenient but, as far as I can see, not strictly necessary.

A good test for any I/O system is whether it can support an Emacs-like screen editor. If your favorite language cannot do it, it can probably appeal to the operating system. If your operating system cannot do it, it is probably not worth using anyway. Conversely, if your language or operating system can support a screen editor, then there is a good chance that they can implement this proposal.

Here is a non-exhaustive list of systems and languages on which this proposal is certainly implementable:

C/Unix	(VAX, Perq, 68000's, etc.)
VAX/VMS	
Multics	
Apollo AEGIS	
Lisp	(Franz Lisp, Common Lisp, Lisp-Machine Lisp, etc.)

The proposed streams are particularly similar to Unix's pipes and Common Lisp's bidirectional stream.

Stream Primitives

Stream primitives can be divided into two groups: stream creation and stream actions. Streams can be created in a variety of ways (from files, processes and devices), but once created they admit the same set of I/O operations.

Stream creation primitives are:

---

(3) Suggested by Mark Manasse

```
stream      : unit -> stream
file        : string -> stream
save        : string -> stream -> unit
channel     : string -> stream
terminal    : stream
```

stream creates a new empty stream at every invocation.

file creates a stream out of a file (the argument is the file name). The file is not affected by this operation, nor by further operations on the stream(s) extracted from it (except "save"). If the file does not exist, an empty one is created.

save stores the *current* contents of a stream on a file (the first argument is a filename). The stream is not affected; any preexisting file having that filename is deleted. Because of the way "file" operates, there is no distinction between empty and non-existing files. Hence a file can be erased by a "save" operation on it with an empty stream as argument.

channel creates a new stream which is a channel to the process identified by its argument. The argument (which is operating system dependent) will usually describe an executable program which is activated as a separate process with its input and output connected to the stream.

terminal is the standard terminal channel: input form this channel will read from the user terminal, and output will print to the user terminal. Auxiliary terminals and other devices can be linked by specialized channel commands. The semantics of the terminal *device* is not specified in detail, to accommodate different kinds of terminals and protocols. Moreover, terminal echoing and buffering are local properties of the device, not of the terminal channel, and are not part of the semantics of the I/O operations. Properties of the terminal device can conceivable be changed when needed by communication to the operating system through an auxiliary channel.

Stream action primitives are:

```
input       : stream -> int -> string
output      : stream -> string -> unit
lookahead   : stream -> (int # int) -> string
caninput    : stream -> int -> bool
```

input inputs n characters from a stream. Input is *unbuffered* (i.e. the characters are available as soon as the process at the other end produces them) and *uninterpreted* (i.e. they may include backspaces, line dels, etc.: all the editing characters normally interpreted by the operating system(4)). The effect of buffering can be achieved by n-char "output" operations at the other end of the stream, and n-char "input" operations at this end. Interpretation must be programmed. If not all the n characters are currently available, the operation *hangs* until they can be read from the stream. The n characters are then *extracted* from the stream and are no longer available to succeeding input operations.

output outputs a string to a stream. Output is *unbuffered* (all the characters are immediately available at the other end of the stream) and *uninterpreted* (all Ascii characters can be transmitted on a stream). Buffering can be achieved by emitting long strings (this is normally going to be more efficient than emitting one character at a time). Interpretation of special signals (e.g. a character meaning "this is the end of the message") has to be done at the other end of the stream. An output

(4) The terminal process normally interprets editing characters locally, and sends out one line at a time. In this case line editing happens even if the stream is uninterpreted. In a different mode (needed for example in screen editors) the terminal process transmits immediately every key stroke. In this case the uninterpreted stream transmits the characters unchanged to the other end, where interpretation must be programmed.

operation *inserts* characters in a stream: any succeeding "output" operation will insert characters after those.

lookahead behaves very much like "input", but does not affect the stream. Given two integers, it returns a segment of a stream: the first integer is the starting position of the segment in the stream (the first character is at position 1), and the second integer is the length of the segment. Like "input", it hangs until enough characters are available. The efficiency of lookahead should be expected to deteriorate for long or remote segment.

caninput determines whether n character are *currently* available for read on a stream: if they are not, it returns false without hanging. The normal use is "if caninput s n then input s n" or "if caninput s (n+m-1) then lookahead s (n,m)" when it is undesirable to hang.

All the above primitives can rise exceptions because of I/O errors, failing with their respective names.

Examples

Here is how to copy a stream to another stream:

```
val CopyStream (InStream: stream, OutStream: stream) : unit =
    while caninput InStream 1 do
        output OutStream (input InStream 1);
```

Without realizing it, we have just written a program which can print a file to the terminal:

```
val PrintFile (FileName: string) : unit =
    CopyStream(file FileName, terminal);
```

Using CopyStream in the "opposite" direction, we can write a program to append memos to the end of (possibly preexisting) memo files; the memo is read from the terminal, and is terminated by the break key which produces an "interrupt" exception:

```
val Memo (FileName: string) : unit =
    let val MemoStream = file FileName
    in (CopyStream(terminal, MemoStream)
        ? save FileName MemoStream)
```

And here is a program which prints the date (under Unix):

```
val Date () =
    CopyStream(channel "/bin/date", terminal);
```

Input prompts can be programmed this way: Prompting_InChar is a version of InChar (i.e. of "input s 1"), which prompts for input.

```
local val lastchar = ref ""
in val Prompting_InChar (stream: stream) (prompt: string) : string =
    (if !lastchar = "\L"
     then output stream prompt
     else ();
     lastchar := input stream 1;
     !lastchar);
```

Note that, because of bipolar streams, we do not need to pass two streams to Prompting_InChar.

Notes

Multiplexing. A stream is said to be input (output) multiplexed when it is used by different parts of a program for conceptually distinct purposes. Multiplexing is possible and well defined. Multiplexed input and output operations simply interleave; any of those operations may affect the result of all the succeeding I/O operations in all the other parts of the program. The predefined *terminal* stream is multiplexed between the user and the ML system: both can do I/O on the user terminal. This multiplexing is set up so that every DEL char (Ascii 127, but this may depend on implementations) typed at the keyboard is captured by the ML system and never reaches the user program. The ML system uses this character to induce an "interrupt" exception in the user program, so that computations can be interrupted even when they are on wait or input-loop on the terminal input.

Device Mapping. Initially the "terminal" *stream* is connected to the standard "terminal" *device*. The terminal stream can be redirected to other devices, files or processes by operating-system commands. However this should rarely be necessary: the correct practice is to parameterize every program with respect to the streams it uses, so that we can pass it "terminal" or some other stream.

ML-to-ML communications. These should happen through auxiliary channels (i.e. not through "terminal" streams). The protocol to establish these channels may require running ad-hoc external processes, or communicating directly with the operating system; hence it is not described here.

Dead channels. If the process at the other end of a channel dies for some reason, and if its death can be detected, then all the operations on that channel should fail. When dead channels cannot be detected, they look just like empty streams.

Saving channels. "save" can be used to write the current contents of an external stream (i.e. the portion which has been already generated by the other side of the channel but not yet absorbed by this side) on a file, without affecting the stream.

Stream lifetime. Only the garbage collector can eliminate streams when they are no longer needed. If that stream is a channel, after the elimination it will look like a dead channel to processes at the other end. Operating systems usually allow only a very small number of files and channels to be active at the same time. If a larger number of streams are generated, and cannot be garbage collected, some of the old files or channels can be temporarily deactivated, until needed again. Deactivate channels are not dead.

Streams never end. There is no way of telling whether a file or a channel is "finished": more input may come at any moment. Still, we want to be able to test for empty stream at any particular moment. The only way to do this is to see whether it is possible to read one more character: at some level this implies a real read operation on the stream. This is why instead of an *emptystream* primitive (which can be defined as "caninput s 1") we have the more explicit *caninput*. Moreover, an n-char caninput is needed in conjunction with n-char input and lookahead operations. Caninput never hangs: it always immediately returns the current state of the stream.

Streams never say ouch! Some devices (typically terminals) may require the transmission of full 8-bit characters to execute special functions, as opposed to the Ascii 7-significant-bits characters. Hence a character should be intended as an 8-bit quantity. Characters with the 8th bit high should be denoted inside strings by a new escape sequence "\#c", where "c" is any of the old characters or escape sequences, e.g. "\#L" (high L), "\#\L" (high linefeed), "\#\^C" (high control-C), etc. Any 8-bit character can be transmitted on a stream; no exceptions. Some systems use particular characters for particular functions (e.g. end-of-transmission), or use the 8th bit for parity checking. In these situations all the anomalous characters have to be transparently encoded and decoded to transmit them on streams. When reading a stream from the terminal, it may be necessary to have an end-of-stream character to terminate the input (e.g. ^D in Unix). This must be explicitly programmed. Note that it may be sufficient to use the break key and trap the corresponding exception.

The power of lookahead. "lookahead" may seem too general. Here is an application where all its power is needed. Suppose one wants to write an input scanner which recognizes arbitrary

regular expressions (these programs actually exist under Unix and are widely used). After recognizing an initial segment of the input, one wants to look ahead a few more characters (without starting from scratch again); a failure to recognize those characters might require to backtrack before the current position. Hence one wants to "lookahead" arbitrary segments of the input, and actually commit himself to "input" only when everything is settled. An alternative could be to have and "uninput" primitive, replacing "lookahead", which puts character back on the stream so that they can be "input" again. Unfortunately "uninput" interacts badly with multiplexing.

Substandards. Robin Milner suggests the following substandard to facilitate implementations. All internal streams are unipolar; "stream" creates unipolar output streams, "file" creates unipolar input streams. "input s 0" and "output s "" " can be used to test the polarity of a stream (because of the exceptions they may generate) and to determine whether an implementation is substandard. Another substandard can consist in implementing a "peek s n" primitive, equivalent to "lookahead s (1,n)", instead of the full "lookahead".

Operating system requests. Operating-system dependent operations can be performed by communication with the operating system itself. This can be done by opening a channel to an operating-system command interpreter. For example, under Unix the current date could be obtained by opening a channel to the "date" process (val DateProcess = channel"/bin/date"; val Date = ReadOneLine DateProcess;), or by opening a channel to a shell, and asking it to execute the "date" command (val ShellProcess = channel"/bin/sh"; output ShellProcess "date val Date = ReadOneLine ShellProcess). Note that this practice is potentially dangerous; it should be used only in cases of extreme necessity, and until we agree on new language primitives which can do the same job.

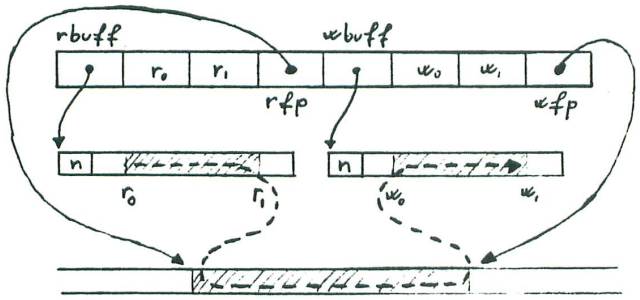
File Protection. If a file is write-protected (e.g. system files), then a "file" operation on it will open an unipolar stream of the appropriate polarity. Similarly, if a file is read-protected. If a file is both read and write protected, then a "no-polarity" stream should be opened, on which both input and output operations fail. Protection schemes seem to be too operating-system dependent to be included in this proposal. Under Unix the protection of a file could be changed by opening a channel to a shell process, and instructing it to change the protection by a Unix command.

Are streams fast? In I/O, speed is only a factor of the implementation effort (given some basic raw speed of the storage devices), because there are so many things that can be optimized. I am confident that the stream primitives can approach the raw speed of the operating system primitives within sensible bounds. However this could be compromised when the operating system gets in the way by not providing simple and efficient character-level I/O primitives.

Implementation notes

These are ideas about the implementation of streams in Unix-like environments. The critical assumption seems to be the presence of multiple file descriptors pointing to the same file. I think that, in this restricted context, multiple descriptors can be simulated in environments which only provide unique descriptors (at some cost), but I do not have very clear ideas about this problem.

Internal Streams. Internal streams are the ones obtained by "file" and "stream". They should be kept in memory as far as possible, and all movements to disk should happen in large chunks. A way of doing this is to use two circular buffers and two file descriptors for every internal stream. The file descriptors are only allocated at the first read and the first write operation respectively, and not at the time of creation of the stream. The semantics of "file" requires that the file originating the stream be copied before any write operation on the stream; the copy should not be made as long as simple read operations are performed.



If the stream contents are shorter than the sum of the lengths of the read and write buffers, everything is kept in memory. When the buffers underflow or overflow, data is read form, or written to the file in buffer units. Note that internal stream I/O is buffered even if the I/O primitives look unbuffered to the user!

The stream primitives can be based on the following InChar and OutChar routine sketches (referring to the previous figure), where +(n) is sum modulo n, and n is the size of the buffers:

```
OutChar(c) =
      if w1 +(n) 1 = w0
      then WriteBufferFull(c)
      else (wbuffer[w1] := c;  w1 := w1 +(n) 1)

WriteBufferFull(c) =
      if rfp = wfp  {file is empty}
      then  if r1 +(n) 1 = r0
            then (WriteBufferToFile();  wbuffer[w1] := c;  w1 := w1 +(n) 1)
            else (rbuffer[r1] := c;  r1 := r1 +(n) 1)
      else (WriteBufferToFile();  wbuffer[w1] := c;  w1 := w1 +(n) 1)

InChar() =
      if r0 = r1
      then ReadBufferEmpty
      else (result := rbuffer[r0];  r0 := r0 +(n) 1;  result)

ReadBufferEmpty() =
      if rfp = wfp  {file is empty}
      then  if w0 +(n) 1 = w1
            then Hang()
            else (result := wbuffer[w0];  w0 := w0 +(n) 1;  result)
```

```
else (ReadBufferFromFile(); result := rbuffer[r0]; r0 := r0 + (n) 1; result)
```

The "save" primitive may produce unnecessary copying of files, e.g. when a file is saved for the last time it would be sufficient to rename the temporary file implementing the stream instead of copying it. Instead of changing the semantics of save (I don't see any way of doing it which would not require unnecessary copying in some situation), I think one should implement a "copy-by-need" strategy. For example, "save" could always rename the temporary file to the real file without copying, but the first time the stream is subsequently affected, the real file is copied back into a temporary stream file.

External streams. External streams are "terminal" and the ones obtained by "channel". They are unbuffered, but they still need an unbounded input buffer to implement the lookahead and can-input operations. This unbounded input buffer can be implemented as a fixed length in-memory buffer with an overflow file on disk, if needed. The file descriptors here are replaced by device or channel descriptors.