Proposed interface for Standard ML Stream I/O

Andrew W. Appel

September 26, 1994

1 Introduction

Desirable properties:

- Buffering
- Efficiency
  - Minimize number of system calls
  - Minimize memory-memory copying
  - Short normal-case control paths
- "Lazy functional stream" interface, for clean and flexible lookahead.
- Layering: allow "buffered I/O system" to be layered on any primitive unbuffered "reader" or "writer."
- Parametrization with respect to character-streams or byte-streams, etc.
- Optional buffering: allow the unbuffered I/O through the "buffered stream" interface.
- Allow non-blocking reads (writes?).

2 Organization

Two levels: **PRIM_IO** describes "file descriptor, system call" input/output. **STREAM_IO** describes buffered stream input output. The Stream I/O system is a functor:

> **functor StreamIO(structure PrimIO : PRIM_IO ...** *etc*) **: STREAM_IO**

3 PRIM_IO

Primitive I/O is at the level of file descriptors and system calls.

```
signature PRIM_IO =
sig
    type elem
    type vector
    type array

    exception Io of {
        ml_op    : string,
        name     : string,
        os_op    : string,
        reason   : string,
        syserror : OS.syserror
      }

    datatype writer = Wr of
              {write_noblock: (vector -> int option) option,
               writea_noblock: (array*int*int-> int option) option,
               write_block: (vector -> int) option,
               writea_block: (array*int*int -> int) option,
               block: (unit->unit) option,
               can_output: (unit->bool) option,
               name: string,
               chunksize: int,
               close: unit -> unit,
               tell : (unit->int) option,
               seek : (int->unit) option,
               file : OS.file option}



    datatype reader = Rd of
              {close    : unit -> unit,
               name: string,
               chunksize: int,
               read_noblock : (int -> vector option) option,
               read_block :   (int -> vector) option,
               reada_noblock: (array*int*int->int option) option,
               reada_block:   (array*int*int->int) option,
               block    : (unit -> unit) option,
               can_input: (unit -> bool) option,
```

```
             tell     : (unit -> int) option,
             seek     : (int -> unit) option,
             size     : unit -> int,
             file : OS.file option}
             }

    val open_in: string -> rd
    val open_out: string -> wr

end
```

A file (device, etc.) is a sequence of "elements" (**elem**), which may (for example) be characters or bytes. The distinction between characters and bytes is necessary on DOS, where CR-LF is translated to LF when reading character files; or on Windows-NT where characters are 16-bits (Unicode) and bytes are 8 bits.

One typically reads or writes a sequence of elements in one system call: this sequence is the **vector** type. Sometimes it is useful to write the sequence from a mutable **array** instead of from the vector.

A **reader** is a file (device, etc.) opened for reading, and a **writer** one opened for writing.

The components of a **writer** are:

write_noblock(v)
: This (optional) function without blocking writes elements $v_0, \ldots, v_{k-1}$, for $k \leq n$ to the output device, and returns SOME(k); or (if the write would block) returns NONE. $k = 0$ is not recommended (prohibited?).

writea_noblock(a,i,n)
: This (optional) function without blocking writes elements $a_i, \ldots, a_{i+k-1}$, for $k \leq n$ to the output device, and returns SOME(k); or (if the write would block) returns NONE. $k = 0$ is not recommended (prohibited?).

write_block(v)
: This (optional) function writes elements $v_0, \ldots, v_{k-1}$, for $0 < k \leq n$ to the output device, and returns k. If necessary, waits (blocks) until the external world can accept at least one element.

writea_block(a,i,n)
: This (optional) function without blocking writes elements $a_i, \ldots, a_{i+k-1}$, for $0 < k \leq n$ to the output device, and returns SOME(k); or (if the write would block) returns NONE. If necessary, waits (blocks) until the external world can accept at least one element.

block()
: This (optional) function does not return until the writer is guaranteed to be able to write without blocking.

can_output()
: (optional) Returns **true** iff the next write can proceed without blocking.

name
: The name associated with this file or device, for use in error messages shown to the user.

chunksize
: The recommended (efficient) size of write operations on this writer. This is typically to the block size of the operating system's buffers. If that is not known, a value of 2048 or 4096 will probably work well. **Chunksize** = 1 strongly recommends (but cannot guarantee, since buffering occurs in other modules, not this one) unbuffered I/O on the writer. **Chunksize** = 0 is illegal.

close()
: Closes the writer (for example, frees operating system resources devoted to this writer). Further operations to this writer are illegal (but need not be checked for(?)).

tell()
: (optional) Tells the number of elements in the file between the beginning and the current position. (Initially, **tell**() = 0.) Useful only on seekable writers.

seek(i)
: (optional) Moves to position i in the file, so future writes occur at this position.

file
: (optional) The operating-system file descriptor associated with this file.

One of **write_block** or **write_noblock** must be provided. Providing more of the optional functions increases functionality and/or efficiency of clients:

1. Absence of both **write_block** and **block** means that blocking output is not possible.

2. Absence of both **write_noblock** and **can_output** means that non-blocking output is not possible.

3. Absence of **write_noblock** means that non-blocking output requires two system calls (using **can_output, write_block**).

4. Absence of **writea_block** or **writea_nonblock** means that extra copying will be required to write from an array.

5. Absence of **tell** means that buffered seeking may be less efficient.

6. Absence of **seek** prevents random access.

The components of a **reader** are

close()
: Closes the reader (for example, frees operating system resources). Further operations to this writer are illegal (but need not be checked for(?)).

name
: The name associated with this file or device, for use in error messages shown to the user.

chunksize
: The recommended (efficient) size of read operations on this reader. This is typically to the block size of the operating system's buffers. If that is not known, a value of 2048 or 4096 will probably work well. **Chunksize** = 1 strongly recommends (but cannot guarantee, since buffering occurs in other modules, not this one) unbuffered I/O on this reader. **Chunksize** = 0 is illegal.

read_noblock(n)
: (optional) Reads i elements without blocking, for $0 < i \leq n$ creating a vector v, returning SOME(v); or (if a read would block) returns NONE.

read_block(n)
: (optional) Reads i elements for $0 < i \leq n$ returning a vector v; blocks (waits) if necessary until at least one element is available.

block()
: (optional) Returns only when at least one element is available for read without blocking.

can_input()
: (optional) Returns **true** iff the next read can proceed without blocking.

tell()
: (optional) Tells the current position in the file (0 means beginning of file). Useful even for non-seekable files, if the **size** function is provided (because large input operations are more efficient if the distance from "here to end of file" is known).

seek(i)
: (optional) Move to position i in file.

size()
: Hint at the approximate total size (number of elements) of the file. If it is inconvenient to support **size** accurately, gross inaccuracy (even to the extent of always reporting 0) is permitted.

file
: (optional) The operating-system file descriptor associated with this file.

One of **read_block** or **read_noblock** must be provided. Providing more of the optional functions increases functionality and/or efficiency of clients:

1. Absence of both **read_block** and **block** means that blocking input is not possible.

2. Absence of both **read_noblock** and **can_input** means that non-blocking input is not possible.

3. Absence of **read_noblock** means that non-blocking input requires two system calls (using **can_input, read_block**).

4. Absence of **reada_noblock** or **reada_block** means that input into an array requires extra copying. *But I do not anticipate that reading into arrays will normally be very important in the "lazy functional stream" model.*

5. Inaccuracy of **size** means that very large inputs (where vectors must be pre-allocated) cannot be done efficiently (in one system call, without copying) if **size** is reported too small, or will cause excess memory allocation if **size** is reported too large. Recommendation: `size=fn()=>0` is acceptable; an approximately accurate **size** is better; an accurate **size** is best.

6. Absence of **tell**, in the unusual case where a buffered system is applied to a reader *not positioned at the beginning of the file*, may lead to excessive memory allocation of vectors for very large input operations.

7. Absence of **tell** means that buffered seeking may be less efficient.

8. Absence of **seek** prevents random access.

Any of the component functions of readers or writers may raise the **Io** exception. No other exceptions should be raised. The components of **Io** are:

ml_op
: The name of the reader/writer component function raising the exception.

name
: Should equal the **name** component of the reader or writer.

os_op
: The name of the operating system call (if any) that failed, otherwise empty.

syserror
: If the Io exception is raised as the result of handling an **OS.SysErr** exception, then the **reason** code provided by the operating system. Otherwise, **OS.noError**.

reason

If **syserror** $\neq$ **OS.noError**, then **OS.errorName(syserror)**; otherwise, a textual summary of the error.

The functions **open_in** and **open_out** provide system-default ways to create readers from "file names." Structures matching this signature may leave these two functions unimplemented (by having them raise the **Io** exception) if there is no appropriate system default.

4 STREAM_IO

The Stream I/O interface provides buffered reading and writing to input and output streams.

Input streams are treated in the lazy functional style: that is, input from a stream f yields a finite vector of elements, plus a new stream f'. Input from f again will yield the same elements (or a prefix or extension thereof); to advance within the stream in the usual way it is necessary to do further input from f'. This interface allows arbitrary lookahead to be done very cleanly, which should be useful both for *ad hoc* lexical analysis and for table-driven, regular-expression-based lexing.

Output streams are handled more conventionally, since the lazy functional style doesn't seem to make sense for output.

Users not interested in lazy functional streams should see section 6.

```
signature STREAM_IO =
sig
    structure PrimIO: PRIM_IO

    type elem   sharing type elem = PrimIO.elem
    type vector sharing type vector = PrimIO.vector

    type instream
    type outstream

    val open_in : string -> instream
    val mk_instream  : PrimIO.reader * string -> instream
    val close_in      : instream -> unit
    val seek_in       : instream * int -> instream
    val tell_in       : instream -> int
    val input         : instream -> vector * instream
    val input_all     : instream -> vector
    val input_noblock : instream -> (vector * instream) option
    val input1        : instream -> elem option * instream
    val input_n       : instream * int -> vector * instream
```

```
    val end_of_stream : instream -> bool
    val get_reader    : instream -> PrimIO.reader

    val open_out: string -> outstream
    val mk_outstream : PrimIO.writer * string -> outstream
    val close_out : outstream -> unit
    val output    : (outstream * vector) -> unit
    val output1   : (outstream * elem) -> unit
    val flush_out : outstream -> unit
    val tell_out  : outstream -> int
    val seek_out  : outstream * int -> unit
    val get_writer: outstream -> PrimIO.writer

end
```

Each instream f can be viewed as a sequence of "available" elements (the buffer or sequence of buffers) and a mechanism (the **reader**) for obtaining more. After an operation $v = \textbf{input}(f)$ it is guaranteed that v is a prefix of the available elements. In a "truncated" instream, there is no mechanism for obtaining more, so the "available" elements comprise the entire stream. In a "terminated" outstream, there is no mechanism for outputting more, so any output operations will raise the **Io** exception.

PrimIO
Every instance of STREAM_IO is built over an instance of PRIM_IO.

elem
A single element (member of a stream).

vector
A sequence of elements, just as in PRIM_IO.

$f =$ **open_in**(s)
Opens a file named s as a stream f. "Default" implementations of STREAM_IO will support **open_in**; other implementations may choose to support only **mk_instream**, raising **Io** on **open_in**.

$f =$ **mk_instream**(r, s)
Create a buffered stream f from a reader r. For purposes of identifying f to the user if exceptions occur, use the name s. (Most users will normally use **open_in** instead.)

close_in(f)
Truncate f, and release operating system resources associated with the underlying file (if any).

$g = $ **seek_in**(f, i)

Now g is a new instream starting from position i of f. f may or may not be truncated. (Nondeterministic behavior! is that bad?) *Not always supported.*

tell_in(f)

Return the current position (elements since beginning of file, starting at 0) of f. *Not always supported.*

$v = $ **input**(f)

If any elements of f are available, return sequence v of one or more elements. If f is at end of file, return the empty sequence. Otherwise read from the operating system (which may block) until one of those conditions occurs.

$v = $ **input_all**(f)

Return the vector v of all the elements of f up to end of stream is reached.

input_noblock(f)

If any non-empty sequence v of f is available or can be read from the operating system without blocking, return SOME(w) where w is any non-empty prefix of v. Otherwise return NONE.

input1(f)

If at least one element e of f is available, return SOME(e). If f is at end of file, return the NONE. Otherwise read from the operating system (which may block) until one of those conditions occurs. Perfectly simulated by:

```
fun input1(f) = let val v = input f
                 in if size(v)=0 then NONE else SOME(sub(v,0))
                end
```

input_n(f, n)

If at least n elements remain before end of stream, return the first n elements. Otherwise, return the (possibly empty) sequence of elements remaining before end of stream. (This was the behavior of the **input** function in the 1989 *Definition of Standard ML.*)

end_of_stream(f)

True if any characters are available in f; false if f is at end of stream. Otherwise reads (perhaps blocking) until one of these conditions occurs. Exactly equivalent to `(size(input f)>0)`.

get_reader(f)

Extract the underlying **reader** from f. Truncates f. Careful users should probably do something like

```
let val r = get_reader f
    val v = input_all f
 in ...
end
```

so as to obtain the elements v already in the buffer before doing anything with r.

$f =$ **open_out**(s)
Open (for writing) a file named s (creating it if necessary) as an outstream f. *Not always supported.*

$f =$ **mk_outstream**(w, s)
Create a buffered outstream f from a writer w. For purposes of identifying f to the user if exceptions occur, use the name s.

close_out(f)
Flush f's buffer, terminate f, then close the underlying writer (releasing operating-system resources associated with it).

flush_out(f)
Flush f's buffer: that is, make the underlying file reflect any previous **output** operations.

output(f, v)
Write the sequence v to f.

output1(f, x)
Write the element x to f.

get_writer(f)
Get the underlying writer associated with f. Flushes and terminates f.

tell_out(f)
Give the current position of f in the underlying file. *Not always supported.*

seek_out(f, i)
Set the current position of f in the underlying file to i. Flush f if necessary. *Not always supported.*

Any prefix of the concatenation of previous writes (since the last seek or flush) may be reflected in the underlying file.

Operations marked *Not always supported* may fail on some streams or in some instiations of the STREAM_IO signature, raising **Io**$\{$syserror = OS.noError, $\ldots\}$. (Should we make a special OS.notSupported?)

Rules: The following expressions are all guaranteed **true**, if they complete without exception.

```
let val a = input f
    val _ = close_in f
    val b = input f
 in  a=b
end
```

```
let val a = input f   in  can_input f   end
```

Note that **can_input** does not imply that more characters remain before end-of-file, just that reading won't block.

```
(close f; can_input f)
```

```
let val a = input_all f   in  end_of_stream f   end
```

```
let val a = input f   in  (size(a)=0) = (end_of_stream f)   end
```

5 StreamIO

The functor **StreamIO** layers a buffering system on a primitive IO module:

```
functor StreamIO(structure PrimIO : PRIM_IO
                 structure Vec: MONO_VECTOR
                 structure Arr: MONO_ARRAY
                 val some_elem : PrimIO.elem
               sharing type PrimIO.elem = Arr.elem = Vec.elem
               sharing type PrimIO.vector=Arr.vector=Vec.vector
               sharing type PrimIO.array=Arr.array
               ) : STREAM_IO = ...
```

The **Vec** and **Arr** structures provide Vector and Array operations for manipulating the vectors and arrays used in PrimIO and StreamIO. The element **some_elem** is used to initialize buffer arrays; any element will do.

Implementation notes:

The previous section gives the specification of **StreamIO** behavior.

Here are some suggestions for efficient performance:

- Operations on the underlying readers and writers (**read_block**, etc.) are expected to be expensive (involving a system call, with context switch).
- Small input operations can be done from a buffer; the **read_block** or **read_nonblock** operation of the underlying reader can replenish the buffer when necessary.

- Keep the position of the beginning of the buffer on a multiple-of-**chunksize** boundary, and do **read** or **write** operations with a multiple-of-**chunksize** number of elements.

- For very large **input_all** or **input_n** operations, it is (somewhat) inefficient to read one **chunksize** at a time and then concatenate all the results together. Instead, it is good to try to do the read all in one large system call; that is, **read_block**(n). However, in a typical implementation of **read_block** this requires pre-allocating a vector of size n. If the user does **input_all**() or **input_n**(maxint), either the size of the vector is not known *a priori* or the allocation of a much-too-large buffer is wasteful. Therefore, for large input operations, query the size of the reader using **size**, subtract the current position, and try to **read** that much. But one should also keep things rounded to the nearest **chunksize**.

  Since **size** is permitted to be inaccurate—in particular, some implementations may just return 0—something reasonable should be done in any case.

- Similar suggestions apply to very large **output** operations. Small outputs go through a buffer; the buffer is written with **writea_block**. Very large outputs can be written directly from the argument string using **write_block**.

- But how should the current buffer position be remembered? Either a **tell** every time **size** is called, or a **tell** when **mk_instream** is first called, followed by careful maintenance of the position of the beginning of the buffer. (Remember, **mk_instream** might be called only after the underlying reader has been moved away from the beginning position.)

- A lazy function instream can (should) be implemented as a sequence of immutable (vector) buffers, each with a mutable ref to the next "thing," which is either another buffer, the underlying reader, or an indication that the stream has been truncated.

- The **input** function should return the largest sequence that is most convenient; usually this means "the remaining contents of the current buffer."

- To support non-blocking input, use **read_noblock** if it exists, otherwise do **can_input** followed (if appropriate) by **read_block**.

- To support blocking input, use **read_block** if it exists, otherwise do **read_noblock** followed (if would block) by **block** and then another **read_noblock**.

- To support lazy functional streams, **reada_block** and **reada_noblock** are not useful; they are included only for completeness.

- **Seek_in**, if seeking forward, might choose to follow the buffer sequence, and can perhaps satisfy the **seek** request without any underlying reader operation.
- **Tell_in**, in some implementations, can tell the position without a system call, if it knows the position of the beginning of the buffer and the current position within the buffer.
- **writea_block** should, if necessary, be synthesized from **write_block**, and vice versa. Similarly for **writea_noblock** and **write_noblock**; **reada_noblock** and **read_noblock**; **reada_block** and **read_block**.

6 Side-effecting instreams

Users not interested in the "lazy functional" model of input can use the **IO** interface, which is a more conventional model. However, **IO** is created by *explicitly* (not abstractly) wrapping a **ref** around a lazy functional stream. This means that one can always apply the STREAM_IO operations to IO streams when desired.

```
signature IO =
  sig
      structure StreamIO : STREAM_IO
      type instream (* = S.instream ref *)
      type outstream (* = S.outstream ref *)
      type elem     sharing type elem = StreamIO.elem
      type vector   sharing type vector = StreamIO.vector

      val open_in       : string -> instream

      val close_in      : instream -> unit
      val seek_in       : instream * int -> unit (* may raise Io *)
      val tell_in       : instream -> int  (* always succeeds *)
      val input         : instream -> vector
      val input_all     : instream -> vector
      val input_noblock : instream -> vector option
      val input1        : instream -> elem option
      val input_n       : instream * int -> vector
      val end_of_stream : instream -> bool
      val lookahead     : instream -> elem option

      val open_out  : string -> outstream
      val close_out : outstream -> unit
      val output    : (outstream * vector) -> unit
```

```
    val output1    : outstream * elem -> unit
    val tell_out  : outstream -> int
    val seek_out  : outstream * int -> unit
    val flush_out : outstream -> unit
end
```

The input operations are just as before, except that instead of returning the new lazy function stream they just update the ref with it.

The output operations (which were not lazy functional to begin with) are even more similar in STREAM_IO and IO. The only purpose of the extra **ref** in IO is to allow "output redirection."