Proposed interface for Standard ML Stream I/O

Andrew W. Appel

November 3, 1994

1 Introduction

The Input/Output interface provides:

- buffered reading and writing;
- arbitrary lookahead, using an underlying “lazy streams” mechanism;
- dynamic redirection of input or output;
- random access;
- uniform interface to text and binary data;
- layering of stream translations, through an underlying “reader/writer” interface;
- unbuffered input/output, through the reader/writer interface or even through the buffered stream interface;
- primitives sufficient to construct facilities for random access reading/writing to the same file.

In addition, the prescriptions and recommendations herein allow for efficient implementation, minimizing system calls and memory-memory copying.

The I/O system has several layers of interface. From bottom to top, they are

PRIM_IO Uniform interface for unbuffered reading and writing at the “system call” level, though not necessarily via actual system calls.

STREAM_IO Buffered “lazy functional stream” input; buffered conventional output.

IO Buffered, conventional (side-effecting) input and output with redirection facility.
Because most programmers will use the IO interface, I will describe that first, rather informally. Then I will go bottom-up over the entire system, giving a technical specification of the interfaces, and their axioms and pragmatics.

## 2 IO

Conventional buffered input/output is done using several structures matching the IO signature: TextIO, for character input/output, BinIO, for binary (byte) input/output.

signature IO =
  sig
  type instream
  type outstream
  type elem
  type vector

  val open_in : string -> instream
  val close_in : instream -> unit
  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 setpos_in : instream * int -> unit (* may raise Io *)
  val getpos_in : instream -> int (* always succeeds *)

  val open_out : string -> outstream
  val close_out : outstream -> unit
  val output : (outstream * vector) -> unit
  val output1 : outstream * elem -> unit
  val flush_out : outstream -> unit
  val getpos_out : outstream -> int
  val setpos_out : outstream * int -> unit

  structure StreamIO : STREAM_IO
  sharing type elem = StreamIO.elem
  sharing type vector = StreamIO.vector

  val mk_instream : StreamIO.instream -> instream
  val get_instream : instream -> StreamIO.instream
val set_instream : instream * StreamIO.instream -> unit
val mk_outstream : StreamIO.outstream -> outstream
val get_outstream : outstream -> StreamIO.outstream
val set_outstream : outstream * StreamIO.outstream -> unit

signature TEXT_IO =
sig
  include IO
  sharing type StreamIO.elem = Word8.word
  sharing type StreamIO.vector = Word8Vector.vector
  val std_in : instream
  val std_out: outstream
  val std_err: outstream
end

signature BIN_IO =
sig
  include IO
  sharing type StreamIO.elem = Word8.word
  sharing type StreamIO.vector = Word8Vector.vector
end

structure TextIO : TEXT_IO
structure BinIO : BIN_IO

Operations on instreams

elem
A single element (member of a stream); for TextIO streams this is char; for BinIO this is Word8.word.

vector
A sequence of elements (such as string or Word8Vector.vector).

\( f = \text{open}_{\text{in}}(s) \)
Opens a file named \( s \) as a stream \( f \).

\( \text{close}_{\text{in}}(f) \)
Close \( f \); no further operations are permitted on \( f \) (they will raise the Io exception).

\( v = \text{input}(f) \)
Read some elements of \( f \), returning a vector \( v \). If (and only if) \( f \) is at end
of file, size(v) = 0. May block (not return until data is available in the external world).

\[ v = \text{input_all}(f)\]

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

\[ \text{input_noblock}(f)\]

If any elements of f can be read without blocking, return at least one of them. If it is possible to determine without blocking that f is at end of stream, return SOME(empty). Otherwise return NONE.

\[ c = \text{input1}(f)\]

If at least one element e of f is available, return SOME(e). If f is at end of file, return the NONE. Otherwise block until one of those conditions occurs.

\[ v = \text{input}_n(f, n)\]

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. Blocks if necessary. (This was the behavior of the input function in the 1989 Definition of Standard ML, and pre-1.00 releases of SML/NJ.)

\[ \text{end_of_stream}(f)\]

False if any characters are available in f; true if f is at end of stream. Otherwise blocks until one of these conditions occurs. Exactly equivalent to (size(input f)=0).

\[ c = \text{lookahead}(f)\]

Return the next character without advancing the stream; or at end of file return NONE. Multiple-character lookahead can be accomplished with the lazy functional stream interface; see section 5.

\[ \text{setpos_in}(f, i)\]

Seek to position i in f.

\[ \text{getpos_in}(f)\]

Tell the current position (elements since beginning of file, starting at 0) of f. Not always supported (raises Io if not supported on f).

Operations on outstreams

\[ f = \text{open_out}(s)\]

Open (for writing) a file named s (creating it if necessary) as an outstream f.
close_out\((f)\)
Flush \(f\)'s buffer and close the stream (releasing operating-system resources
associated with it).

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

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

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

getpos_out\((f)\)
Tell the current position of \(f\) (not always supported).

setpos_out\((f,i)\)
Seek to position \(i\) of \(f\) (not always supported).

There is also a set of primitives to relate IO streams to the “lazy functional streams” model of input/output; and thus to the underlying unbuffered
reader/writer primitives:

StreamIO
The particular instantiation of the STREAM_IO interface underlying
this IO module (i.e., streams of bytes, chars, or some other element type).

\(f = \text{mk_instream}(s)\)
Create a conventional stream \(f\) from a functional stream \(s\).

\(s = \text{get_instream}(f)\)
Extract the functional stream \(s\) from \(f\). This allows arbitrary lookahead;
for example:

fun lookahead_n(f,n) =
  let val f' = mk_instream(get_instream(f))
  in input_n(f',n)
  end

This makes a “copy” \(f'\) of the stream \(f\); then input operations in \(f'\)
won't affect \(f\) (though setpos_in on \(f'\) may effectively close \(f\)). For more
details, see the next few sections.

set_instream\((f,s)\)
Redirect \(f\), so that further input comes from \(s\). For example:
fun from_file(g,name) =  
  let val f = open_in name  
      val save_std_in = get_instream std_in  
  in set_instream(std_in, get_instream f);  
    g();  
    set_instream(std_in, save_std_in)  
  end

For more details, see the next few sections.

\[ f = \text{mk_outstream}(s) \]
Create a conventional outstream \( f \) from a \text{StreamIO.outstream} \( s \). The output streams in \text{StreamIO} are not "functional," they are conventional streams operated on by side-effecting output. The difference between an \text{IO.outstream} and a \text{StreamIO.outstream} is that the former may be redirected using \text{set_outstream}. Think of the former as a ref of the latter.

\[ s = \text{get_outstream}(f) \]
Extract the underlying outstream \( s \) from the redirectable outstream \( f \). Unfortunately, \( s \) is not "pure functional," so there's no equivalent of the lookahead trick shown above. Unlike instreams, if

\[
\text{val } f' = \text{mk_outstream}(\text{get_outstream } f)
\]

then operations on \( f' \) are equivalent to operations on \( f \).

\[ \text{set_outstream}(f, s) \]
Useful for redirecting output. For example,

fun to_file(g,name) =  
  let val f = open_out name  
      val save_std_out = get_outstream std_out  
  in set_outstream(std_out, get_outstream f);  
    g();  
    set_outstream(std_out, save_std_out)  
  end

In can be argued that this is not very elegant; the function \( g \), instead of writing stuff to \text{std_out}, should have been parameterized (in the usual ML way) on an \text{outstream} from the very beginning. Then the \text{get} and \text{set} primitives wouldn't be needed.
3 OS

The primitive I/O (PrimIO), stream I/O (StreamIO), and standard I/O (IO) packages require only these components of the OS structure:

```
structure OS : sig
    type syserror
    val noError : syserror
    exception SysErr of
        {ml_op : string,
         os_op : string,
         reason : syserror}
end
```

All "operating system" operations not listed here (reading, writing, etc.) are parametrized (in the PrimIO.reader and PrimIO.writer types) and may or may not come from the actual operating system.

4 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
        }

        type 'a buf = {
            data : 'a,
            pos : int,
            nelems : int
        }

        datatype writer = Wr of
```

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**\{**buf**=v,**pos**=i,**nelems**=n\}
This (optional) function without blocking writes elements \(v_i, \ldots, v_{i+k-1}\), for \(k \leq n\) to the output device, and returns \texttt{SOME}(k); or (if the write would block) returns \texttt{NONE}. \(k = 0\) is not recommended (prohibited?). Raises \texttt{Io} on failure of underlying system call, or \texttt{Subscript} if \(i < 0\) or \(i + n > \text{length}(v)\).

**writea

**noblock**\{**buf**=a,**pos**=i,**nelems**=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 \texttt{SOME}(k); or (if the write would block) returns \texttt{NONE}. \(k = 0\) is not recommended (prohibited?).

write

**block**\{**buf**=v,**pos**=i,**nelems**=n\}
This (optional) function writes elements \(v_i, \ldots, v_{i+k-1}\), 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**\{**buf**=a,**pos**=i,**nelems**=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 \texttt{SOME}(k); or (if the write would block) returns \texttt{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()
(conditional) Returns \texttt{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. \texttt{Chunksize} = 1 strongly recommends (but cannot guarantee, since buffering occurs in other modules, not this one) unbuffered I/O on the writer. \texttt{Chunksize} \leq 0 is illegal (functions in other modules taking writers as arguments may raise exceptions).

**close()**
Closes the writer (for example, frees operating system resources devoted to this writer). Further operations to this writer are illegal (but it is not the responsibility of the writer to check for this).
getpos()
  (optional) Tells the number of elements in the file between the beginning and the current position. (Initially, getpos() = 0.) Most useful on seekable writers.

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

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_noblock means that extra copying will be required to write from an array.
5. Absence of getpos means that buffered setpos may be less efficient.
6. Absence of setpos prevents random access.

The components of a reader are

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

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 $v$ of length $i$; blocks (waits) if necessary until at least one element is available.

**reada_noblock{buf=a,pos=i,nelems=n}**

(optional) Reads $k$ elements without blocking, for $0 < k \leq n$ into $a_i, \ldots, a_{i+k-1}$, returning **SOME(k)**; if no elements remain before end-of-file, returns **SOME(0)** without blocking; or (if a read would block) returns **NONE**.

**reada_block{buf=a,pos=i,nelems=n}**

(optional) Reads $k$ elements for $0 < k \leq n$ into $a_i, \ldots, a_{i+k-1}$, returning a vector $k$; blocks (waits) if necessary until at least one element is available. If no elements remain before end-of-file, returns 0.

**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.

**getpos()**

(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).

**setpos(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.

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.*

Clients of **PrimIO** are required to synthesize blocking reads from **read_noblock**+**block**, synthesize vector reads from array reads, synthesize array reads from vector reads, as needed—so the **PrimIO.reader** is required to provide only a minimum set. If the **reader** can provide more than the minimum set in *a way that is more efficient then the obvious synthesis* than by all means it should do so. However, providing more than the minimum by just doing (inside the PrimIO layer) the obvious synthesis is not recommended, because then clients won’t get the “hint” about which are the efficient (“recommended”) operations.

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 **getpos**, 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 **getpos** means that buffered **setpos** may be less efficient.

8. Absence of **setpos** 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 \texttt{syserror} \neq \texttt{OS.noError}, then \texttt{OS.errorName(syserror)}; otherwise, a textual summary of the error.

The functions \texttt{open\_in} and \texttt{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 \texttt{Io} exception) if there is no appropriate system default.

5 \textbf{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; 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 \textit{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.

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 setpos\_in : instream * int -> instream

val getpos\_in : instream -> int

val input : instream -> vector * instream

val input\_all : instream -> vector

val input\_noblock : instream -> (vector * instream) option

val input\_1 : 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 getpos_out : outstream -> int
val setpos_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 $(v, f') = \text{input}(f)$ 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.

\[ f = \text{open}_\text{in}(s) \]
Opens a file named $s$ as a stream $f$. “Default” implementations of STREAM.IO will support \text{open}_\text{in}; other implementations may choose to support only \text{mk}_\text{instream}, raising Io on \text{open}_\text{in}.

\[ f = \text{mk}_\text{instream}(r, s) \]
Create a buffered stream $f$ from a reader $r$. For purposes of identifying $f$ to the user if exceptions occur, use the name $s$. In $r$, \text{read}_\text{block}, \text{reada}_\text{block},$ and \text{block} must not all be \text{NONE} or an Io exception will be raised. (Most users will normally use \text{open}_\text{in} instead.)

\[ \text{close}_\text{in}(f) \]
Truncate $f$, and release operating system resources associated with the underlying file (if any).

\[ g = \text{setpos}_\text{in}(f, i) \]
Now $g$ is a new instream starting from position $i$ of $f$. $f$ may or may not
be truncated, depending on whether the setpos request can be satisfied within the buffer. (Nondeterministic behavior! is that bad?) Not always supported.

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

\((u, f') = \text{input}(f)\)
If any elements of \(f\) are available, return sequence \(v\) of one or more elements and the "remainder" \(f'\) of the stream. 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.

\(v = \text{input_all}(f)\)
Return the vector \(v\) of all the elements of \(f\) up to end of stream. Semantically equivalent to:

\[
\begin{align*}
\text{fun input_all}(f) = \text{let val (a, f') = input f} \\
in \text{if size(a) = 0 then a} \\
\text{else a} ^ \text{ input_all } f'
\end{align*}
\]

where ^ is the concatenation operator on element vectors.

\((u, f') = \text{input_noblock}(f)\)
If any non-empty sequence \(v\) of \(f\) is available or can be read from the operating system without blocking, return \(\text{SOME}(w, f')\) where \(w\) is any non-empty prefix of \(v\), and \(f'\) is the "rest" of the stream. Otherwise return \(\text{NONE}\).

\((c, f') = \text{input1}(f)\)
If at least one element \(e\) of \(f\) is available, return \(\text{SOME}(e, f')\). If \(f\) is at end of file, return the \(\text{NONE}\). Otherwise read from the operating system (which may block) until one of those conditions occurs. Semantically equivalent to:

\[
\begin{align*}
\text{fun input1}(f) = \text{let val (v, f') = input f} \\
in \text{if size(v) = 0 then NONE else SOME(sub(v, 0)), f'}
\end{align*}
\]

\((u, f') = \text{input_n}(f, n)\)
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. Blocks if necessary. (This was the behavior of the \texttt{input} function in the 1989 \textit{Definition of Standard ML}.) Semantically equivalent to:

\begin{verbatim}
fun input_n(f,0) = (empty, f)
  | input_n(f,n) = let val (x,f') = input1 f
    val (s,f'') = input_n(f,n-1)
    in (x"s, f'')
  end

end_of_stream(f)
False if any characters are available in \texttt{f}; true if \texttt{f} is at end of stream. Otherwise reads (perhaps blocking) until one of these conditions occurs. Exactly equivalent to \texttt{(size(input f)=0)}.

get_reader(f)
Extract the underlying \texttt{reader} from \texttt{f}. Truncates \texttt{f}. Careful users should probably do something like

\begin{verbatim}
let val r = get_reader f
  val v = input_all f
  in ... end
\end{verbatim}

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

\texttt{f = open_out(s)}
Open (for writing) a file named \texttt{s} (creating it if necessary) as an outstream \texttt{f}. \textit{Not always supported}.

\texttt{f = mk_outstream(w, s)}
Create a buffered outstream \texttt{f} from a writer \texttt{w}. For purposes of identifying \texttt{f} to the user if exceptions occur, use the name \texttt{s}. In \texttt{w}, \texttt{write_block}, \texttt{writea_block}, and \texttt{block} must not all be \texttt{none} or an \texttt{Io} exception will be raised.

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

\texttt{flush_out(f)}
Flush \texttt{f}'s buffer: that is, make the underlying file reflect any previous \texttt{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.

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

setpos_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 setpos or
flush) may be reflected in the underlying file.

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

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

Input is semi-deterministic: input may read any number of elements from
f the "first" time, but then it is committed to its choice, and must return the
same number of elements on subsequent reads from the same point.

let val (a,_,) = input f
    val (b,_,) = input f
in a=b
end

Closing a stream just causes the not-yet-determined part of the stream to
be empty:

let val (a,f') = input f
    val _ = close_in f
    val (b,_,) = input f
in a=b andalso end_of_stream f' 
end (* must be true *)

If a stream has already been at least partly determined, then input cannot
possibly block:

let val a = input f
in case input_noblock f
    of SOME a => a=b
    | NONE => false
end (* must be true *)
Note that a successful \texttt{input\_noblock} does not imply that more characters remain before end-of-file, just that reading won’t block.

Closing a stream guarantees that the underlying reader will never again be accessed; so input can’t possibly block:

\begin{verbatim}
(case (close f; input\_noblock f) of SOME _ => true | NONE => false)
\end{verbatim}

The \texttt{end\_of\_stream} test is equivalent to \texttt{input} returning an empty sequence:

\begin{verbatim}
let val (a,_) = input f in (size(a)=0) = (end\_of\_stream f) end
\end{verbatim}

\textbf{Unbuffered I/O} That is, if chunksize=1 in the underlying reader, then \texttt{input} operations must be unbuffered:

\begin{verbatim}
let val f = mk\_instream(reader)
  val (a,f') = input(f,n)
  val PrimIO.Rd{chunksize,...} = get\_instream f
  in chunksize>1 or else end\_of\_stream f'
end
\end{verbatim}

Though \texttt{input} may perform a \texttt{read}(k) operation on the reader (for \(k \geq 1\)), it must immediately return all the elements it receives. However, this does not hold for partly determined instreams:

\begin{verbatim}
let val f = mk\_instream(reader)
  val _ = do\_input\_operations\_on(f)
  val (a,f') = input(f,n)
  val PrimIO.Rd{chunksize,...} = get\_instream f
  in chunksize>1 or else end\_of\_stream f' (* could be false*)
end
\end{verbatim}

because in this case, the stream \(f\) may have accumulated a history of several responses, and \texttt{input} is required to repeat them one at a time.

Similarly, output operations are unbuffered if chunksize=1 in the underlying writer. Unbuffered output means that the data has been written to the underlying writer by the time \texttt{output} returns.

\textbf{Don’t bother the reader} \texttt{input} must be done without any operation on the underlying reader, whenever it is possible to do so by using elements from the buffer. This is necessary so that repeated calls to \texttt{end\_of\_file} will not make repeated system calls.

This rule could be formalized by defining a “monitor”:

\begin{verbatim}
val monitor: reader -> {rd: reader,
  chars\_read: int ref,
  op\_count: int ref}
\end{verbatim}
and making statements such as:

```ocaml
let val {rd,chars_read,op_count} = monitor(reader)
  val f = mk_instream(rd)
  val (f',n elems) = do_things_counting_elements(f)
  val p1 = getpos_in f'
  val c1 = !chars_read
  val ops = !op_count
  val _ = input f'
  in not ((n elems < c1) andalso (!op_count > ops))
end
```

but perhaps this level of detail is unnecessary.

**Multiple end of file** In Unix, and perhaps in other operating systems, there is no notion of “end of stream.” Instead, by convention a read system call that returns zero bytes is interpreted to mean end of stream. However, the next read to that stream could return more bytes. This situation would arise if, for example,

- the user hits `ctl-D` on an interactive tty stream, and then types more characters;
- input reaches the end of a disk file, but then some other process appends more bytes to the file.

Consequently, the following is *not* guaranteed to be true:

```ocaml
let val z = end_of_stream f
  val (a,f') = input f
  val x = end_of_stream f'
  in x=z (* not necessarily true! *)
end
```

The “don’t bother the reader” rule, combined with the definition of `end_of_stream`, guarantees that

```
end_of_stream(f) = end_of_stream(f).
```

Implementors should beware that an empty buffer sometimes means end of stream, and sometimes not; I found an extra boolean variable necessary to keep track.

## 6 StreamIO

The functor **StreamIO** layers a buffering system on a primitive IO module:
functor StreamIO(structure PrimIO : PRIM_IO
    structure Vec: MONOVECTOR
    structure Arr: MONOARRAY
  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.

If flush_out finds that it can do only a partial write (i.e., writeblock or a similar function returns a "number of elements written" less than its "nelems" argument) then flush_out must adjust its buffer for the items written and then raise an Io exception, in such a way that if the next (or any future) flush_out is successful, no data will have been lost or twice-written.

The same rule applies to output (etc.) if it calls flush_out.

What is the behavior of the Stream_IO primitives if a user interrupt occurs? Reppy thinks that losing information is preferable to printing output twice. This should be cogitated and clarified.

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 `write_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 `getpos` every time `size` is called, or a `getpos` 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.

- `Setpos_in`, if setpos-ing forward, might choose to follow the buffer sequence, and can perhaps satisfy the `setpos` request without any underlying reader operation.

- `Getpos_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`.
7 IO functor

The precise definition of "conventional" streams (IO signature) is in terms of "lazy functional" streams (STREAM_IO). The functor IO is provided:

functor IO(structure S : STREAM_IO) : IO = ...

The structures BinIO and TextIO are (presumably) built using separate applications of this functor (though TextIO is then enhanced with std_in, etc.), but users may apply the StreamIO and IO functors to make streams data types other than char and byte.

The semantics of IO are simple enough that it is sufficient to give a reference implementation.

functor IO(structure S : STREAM_IO) : IO =
let abstraction I =
struct
structure StreamIO = S
  type instream = S.instream ref
  type outstream = S.outstream ref
  type elem = S.elem
  type vector = S.vector
val mk_instream = ref
val get_instream = !
val set_instream = op :=
val mk_outstream = ref
val get_outstream = !
val set_outstream = op :=
val open_in = ref o S.open_in
fun end_of f = if S.end_of_stream f then f else end_of(#2(S.input f))

fun close_in(r as ref f) = (S.close_in f; r := end_of f)
fun setpos_in(r as ref f, i) = r := S.setpos_in(f,i)
val getpos_in = S.getpos_in o !
fun input(r as ref f) = let val (v,f') = S.input f in r:=f'; v end
fun input_all(r as ref f) = let val v = S.input_all f
                              in r := end_of f; v end
fun input_noblock(r as ref f) =
    let val (v,f') = S.input_noblock f in r:=f'; v end
fun input1(r as ref f) = let val (v,f') = S.input1 f in r:=f'; v end
val end_of_stream = S.end_of_stream o !
fun lookahead(ref f) = #1(S.input1 f)

val open_out = ref o S.open_out
val close_out = S.close_out o !
fun output(ref f, v) = S.output(f,v)
fun output1(ref f, x) = S.output1(f,x)
val getpos_out = S.getpos_out o !

fun setpos_out(ref f, i) = S.setpos_out(f,i)
val flush_out = S.flush_out o !
end
in I
end

Note that the `instream` and `outstream` types are abstract.
Some consequences of this definition:
The `end_of_stream` semantics are

fun end_of_stream (f as ref ff) = StreamIO.end_of_stream ff

This implies

let val x = end_of_stream f
    val y = end_of_stream f
in x=y (* guaranteed true *)

Furthermore, second call to `end_of_stream` is guaranteed not to do any system call; this is a consequence of the “Don’t bother the reader” semantics of `StreamIO.input`.
However, reading past end of stream is possible via `input`; the semantics may be straightforwardly derived from the semantics of `StreamIO.input`.
The output operations (which were not lazy functional to begin with) are even more similar between `STREAMIO` and `IO`. The only purpose of the extra `ref` in `IO` is to allow “output redirection.”

8 Random access reading/writing to the same stream

Instreams are instreams, outstreams are outstreams, and ne’er the twain shall meet. At least, not face to face. However, competent users can construct many things from the layered functors.

Here’s an example: reading and writing to the same random-access file without re-opening it.

1. Open the file for reading, and for writing; extract the underlying reader and writer, discarding the buffering layer.

   val reader = TextIO.StreamIO.get_reader (TextIO.StreamIO.open_in name)
   val writer = TextIO.StreamIO.get_writer (TextIO.StreamIO.open_out name)

2. Do some buffered writes; then discard the buffering layer.
let val out = TextIO.mk_outstream(TextIO.StreamIO.mk_outstream(writer,name))
in TextIO.setpos_out(out,some_pos);
  output(out,"Hello ");
  output(out,"World\n");
  flush_out out end

3. Do some buffered reads; then discard the buffering layer.

let val inf = TextIO.mk_instream(TextIO.StreamIO.mk_instream(reader,name))
in TextIO.setpos_in(inf,another_pos);
  input inf;
  input inf end

4. And so on. It's cheap and easy to do mk_instream whenever switching
between reading and writing.

9 Loose ends

What about opening files for append?
  What about user (and other) interrupts during buffered I/O operations?
  Should setpos positions be abstract? How should positions work in translated readers or writers?