Proposed interface for Standard ML Stream I/O

Andrew W. Appel

July 13, 1995

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.

Synopsis

The IO system provides the following signatures, structures, and functors:

```
signature IO
signature PRIM_IO
signature STREAM_IO     (* has a PRIM_IO component *)
signature IMPERATIVE_IO (* has a STREAM_IO component *)
signature TEXT_IO       (* include IMPERATIVE_IO *)
signature BIN_IO        (* include IMPERATIVE_IO *)

structure Position: INTEGER
structure IO: IO
structure TextIO: TEXT_IO
structure BinIO:  BIN_IO

functor PrimIO(...) : PRIM_IO
functor StreamIO(... P: PRIM_IO ...) : STREAM_IO
functor ImperativeIO(... S: STREAM_IO ...) : IMPERATIVE_IO
```

Every implementation must provide the signatures and the structures. The functors are optional, needed only by those users who want to construct buffered I/O systems over element types other than byte and char, or who want non-integer file positions.

2 IMPERATIVE_IO

Conventional buffered input/output is done using structures matching extensions of the **IMPERATIVE_IO** signature: **TextIO**, for character input/output, **BinIO**, for binary (byte) input/output.

```
    signature IMPERATIVE_IO =
      sig
          type instream
          type outstream
          type elem
          type vector
          type subvector
          type pos
          val closeIn      : instream -> unit
          val input         : instream -> vector
          val inputAll     : instream -> vector
          val inputNoBlock : instream -> vector option
          val input1        : instream -> elem option
          val inputN       : instream * int -> vector
          val endOfStream : instream -> bool
          val lookahead     : instream -> elem option
          val setPosIn     : instream * pos -> unit
          val getPosIn     : instream -> pos
          val endPosIn     : instream -> pos
          val closeOut : outstream -> unit
          val output    : (outstream * vector) -> unit
          val outputS    : outstream * subvector -> unit
          val output1    : outstream * elem -> unit
          val flushOut : outstream -> unit
          val getPosOut  : outstream -> pos
          val endPosOut  : outstream -> pos
          val setPosOut  : outstream * pos -> unit
          structure StreamIO : STREAM_IO
          sharing type elem = StreamIO.elem
          sharing type vector = StreamIO.vector
          sharing type subvector = StreamIO.subvector
          sharing type pos = StreamIO.pos
          val mkInstream   : StreamIO.instream -> instream
          val getInstream  : instream -> StreamIO.instream
          val setInstream  : instream * StreamIO.instream -> unit
          val mkOutstream  : StreamIO.outstream -> outstream
          val getOutstream : outstream -> StreamIO.outstream
          val setOutstream : outstream * StreamIO.outstream -> unit
      end

structure Position : INTEGER

    signature BIN_IO =
    sig
         include IMPERATIVE_IO
         sharing type StreamIO.elem=Word8.word
         sharing type StreamIO.vector=Word8Vector.vector
         sharing type pos=FilePosInt.int
         val openIn : string -> instream
         val openOut: string -> outstream
         val openAppend: string -> outstream
    end

    signature TEXT_IO =
      sig
         include IMPERATIVE_IO
         sharing type pos=FilePosInt.int
         structure StreamIO :
            sig include STREAM_IO
                val inputLine: instream -> string
            end
         sharing type StreamIO.elem = char
         sharing type StreamIO.vector = string
         val openIn : string -> instream
         val openOut: string -> outstream
         val openAppend: string -> outstream
         val stdIn : instream
         val stdOut: outstream
         val stdErr: outstream
         val inputLine : instream -> string * instream
      end
```

The redefinition of StreamIO after its definition via include may not be supported in all Standard ML implementations, but the signature TEXT˙IO can be written out in full without the use of include.

```
structure BinIO : BIN_IO
structure TextIO : TEXT_IO
```

These operations may raise the **IO.Io** exception:

```
exception Io of {name: string,
                 function: string,
                 cause: exn}
```

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 = **openIn**(s)
: Opens a file named s as a stream f.

closeIn(f)
: Close f; no further operations are permitted on f (they will raise the **Io** exception).

v = **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 = **inputAll**(f)
: Return the vector v of all the elements of f up to end of stream.

inputNoBlock(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. On streams that do not support non-blocking input, raise General.NonblockingNotSupported.

c = **input1**(f)
: If at least one element e of f is available, return SOME(e). If f is at end of file, return NONE. Otherwise block until one of those conditions occurs.

v = **inputN**(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.)

endOfStream(f)
: False if any elements are available in f; true if f is at end of stream. Otherwise blocks until one of these conditions occurs.

c = **lookahead**(f)
: Return the next element 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 6.

setPosIn(f, i)
: Seek to position i in f. *Not always supported (raises* **Io** *if not supported on f).*[1]

i = **getPosIn**(f)
: Tell the current position of f. Positions may not correspond 1–1 to elements read from the file, but should increase semimonotonically.

: In implementations where bytes in the underlying file do not correspond exactly to characters (or elements) returned by the input operations, it will typically be the case that positions returned by **getPosIn** are counted in terms of the former, not the latter.

: *Not always supported (raises* **Io** *if not supported on f).*

i = **endPosIn**(f)
: Tell the ending position of f. *Not always supported (raises* **Io** *if not supported on f).*

[1] On a pipe or other interactive stream, **setPosIn** will often succeed if "within the buffer" but fail for larger distances. It's difficult for users to write a predicate that tests a stream to see whether random access is supported on the underlying device. In John Reppy's opinion, this is a bug. I've done it this way because otherwise an extra `fstat` system call would be needed on every file, to see whether it supports random access.

Operations on outstreams

closeOut(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.

outputS(f, v)
Write the subsequence (substring) v to f.

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

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

$i =$ **getPosOut**(f)
Tell the current position of f *(not always supported, may raise exception).* Positions may not correspond 1–1 to elements in the file, but should increase semimonotonically.

In implementations where bytes in the underlying file do not correspond exactly to characters (or elements) written by the output operations, it will typically be the case that positions returned by **getPosOut** are counted in terms of the former, not the latter.

$i =$ **endPosOut**(f)
Tell the ending position of f. *Not always supported (raises* **Io** *if not supported on f).*

setPosOut(f, i)
Seek to position i of f *(not always supported, may raise exception).*

Any of these functions may raise the **Io** exception if an operation fails (including **closeOut** if a buffer cannot be flushed). Closing an already closed stream will not cause an exception to be raised.

Random access

In order to avoid unnecessary limitations on file sizes, the **getPos, endPos, setPos** functions all operate on special **Position** integers:

```
structure Position: INTEGER
```

Position.int is abstract and does *not* share with Int.int or with any other integer type.

Users can operate on the **pos** type using **Position.+** and **Position.−**; or (at the risk of being unable to process large files) convert to/from ordinary integers using **Position.toDefault** and **Position.fromDefault**.

Opening files

The IMPERATIVE_IO signature describes operating-system-independent input and output streams. Implementations may provide many ways of creating instreams and outstreams, using network connections, special devices, ML functions that generate or consume elements on the fly, and so on.

But in many contexts a standard way of opening files (named by simple strings)—and standard input, output, and error streams—will suffice. The **TEXT_IO** and **BIN_IO** signatures each include **IMPERATIVE_IO**, plus:

$f =$ **openIn**(s)
Open the file named s for reading.

$f =$ **openOut**(s)
Open the file named s for writing at the beginning, truncating it if it already exists, creating it if not.

$f =$ **openAppend**(s)

Open the file named s for writing *at the end*, creating it if it does not already exist. On Unix and other operating systems that support "atomic append mode," each individual **flushOut** operation (or other output operation that flushes the buffer) appends atomically to the current end of file, even if other processes are appending to the same file between **flushOut** operations. The **openAppend** function opens a file in this mode, if possible.

stdIn

The standard input stream.

stdOut

The standard output stream.

stdErr

The standard stream for writing error messages. It is unbuffered (`flushOut(stdErr)` is not required).

Both **TextIO** and **BinIO** have **stdIn** streams (of different types), but these are implemented on the same underlying file. Users who do buffered input on both **TextIO.stdIn** and **BinIO.stdIn** will see arbitrary interleaving.[2] The treatment of **stdOut** and **stdErr** is analogous.

TEXT_IO

Text streams (**TextIO.instream**) contain lines of text and control characters. Text lines are terminated with `#"\n"` characters.

On operating systems that use **CR-LF** or **CR** as line terminators, these will be translated to single `#"\n"` characters. The inverse translation will be done on output.

More substantial translation will be done on operating systems that use, for example, escape-coded Unicode text files.

The **TextIO** structure provides, in addition to standard functions and values described above,

$s =$ **inputLine**(f)

Read one line from a text file. The terminating newline character (if any) is read from the file but not included in the result string. If end-of-file is reached before a newline character, all characters remaining in the file are returned. Thus, if end-of-file is reached immediately, the empty string will result.

$s =$ **TextIO.StreamIO.inputLine**(f)

Like TextIO.inputLine, but reads from a functional stream, returning a line and a new functional stream.

$f =$ **translateIn**(g)

The default (operating-system specific) translation from binary instreams to text instreams.

$f =$ **translateOut**(g)

The default (operating-system specific) translation from binary outstreams to text outstreams.

Closing files on program exit

All outstreams will be flushed when the ML program exits. Instreams and outstreams may or may not be closed on program exit, depending on conventions of the host operating system.

[2]Reppy prefers that **stdIn**, **stdOut**, **stdErr** be present only in the **TextIO** module, not in **BinIO**; with an alternate method to access the standard input and output file descriptors as buffered binary streams.

Redirecting IO streams

There is also a set of primitives to relate **IMPERATIVE_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 **IMPERATIVE_IO** module (i.e., streams of bytes, chars, or some other element type).

f = **mkInstream**(s)
Create a conventional stream f from a functional stream s.

s = **getInstream**(f)
Extract the functional stream s from f. This allows arbitrary lookahead; for example:

```
fun lookaheadN(f,n) =
  let val f' = mkInstream(getInstream(f))
   in inputN(f',n)
  end
```

This makes a "copy" f' of the stream f; then **input** operations in f' won't affect f (though **setPosIn** on f' may effectively close f). For more details, see sections 4, 5, 6, 7, and 8, which give a more precise specification of stream behavior.

setInstream(f, s)
Redirect f, so that further input comes from s. For example:

```
fun fromFile(g,name) =
 let val f = openIn name
     val saveStdIn = getInstream stdIn
  in setInstream(stdIn,getInstream f);
     g();
     setInstream(stdIn, saveStdIn)
 end
```

For more details, see the next few sections.

f = **mkOutstream**(s)
Create a conventional outstream f from a **StreamIO.outstream** s. The output streams in **StreamIO** are not "functional," they are conventional streams operated on by side-effecting output. The difference between **TextIO.outstream** and **TextIO.StreamIO.outstream** is that the former may be redirected using **setOutstream**. Think of the former as a `ref` of the latter.

s = **getOutstream**(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

```
  val f' = mkOutstream(getOutstream f)
```

then operations on f' are equivalent to operations on f.

setOutstream(f, s)
Useful for redirecting output. For example,

```
fun toFile(g,name) =
 let val f = TextIO.openOut name
     val saveStdOut = getOutstream stdOut
```

```
  in setOutstream(stdOut,getOutstream f);
     g();
     setOutstream(stdOut, saveStdOut)
 end
```

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

3 IO structure

The structure **IO**, matching the signature IO, contains exception constructors common to all the input/output structures and functors.

```
signature IO =
sig
   exception Io of {name: string,
                    function: string,
                    cause: exn}
   exception BlockingNotSupported
   exception NonblockingNotSupported
   exception TerminatedStream
   exception ClosedStream
   val translateRd : BinIO.StreamIO.PrimIO.reader ->
                     TextIO.StreamIO.PrimIO.reader ->
   val translateWr : BinIO.StreamIO.PrimIO.writer ->
                     TextIO.StreamIO.PrimIO.writer ->
end
```

Exceptions

All the IO modules may raise the **Subscript** exception, if given ill-formed array and bounds arguments by a client; or the **Io** exception. In general, when **Io** is raised as a result of a failure in a lower-level module (**PrimIO**), the underlying exception is propagated up as the **cause** component of the **Io** exception value.

This will usually be a **Subscript**, **SysErr**, or **Fail** exception, but the **StreamIO** module will rarely (perhaps never) need to inspect it.

The Io exception (but not other components of this module) is available unqualified at top level. The components of **Io** are:

function
: The name of the **StreamIO** function raising the exception.

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

cause
: The underlying exception raised by the reader or writer, or detected by **StreamIO**. Some of the standard causes are:

- **OS.SysErr** if an actual system call was done and failed;
- **General.BlockingNotSupported** for **output**, **outputS**, **output1**, **flushOut**, if the underlying writer does not support blocking writes; or for **input**, **inputN**, **input1**, **inputAll** if the underlying reader does not support blocking reads.
- **IO.NonblockingNotSupported** for **inputNoBlock**.

- **IO.TerminatedStream** for **setPosIn** on a terminated stream.
- **IO.ClosedStream** for any output operation on a closed file. This exception is actually raised by the reader or writer. (Input operations on closed streams will generally raise **Terminated-Stream**.)

The **cause** field of **Io** is not limited to these particular exceptions. Users who create their own readers or writers may raise any exception they like, which will be reported as the **cause** field of the resulting **Io** exception.

Imperative-IO and StreamIO modules will never raise a bare **TerminatedStream** exception (or Blocking-NotSupported, NonblockingNotSupported, ClosedStream); these exceptions are only used in the *cause* field of the **Io** exception. However, any module may raise **Subscript** directly if given ill-formed arguments, or may raise **Io** with **Subscript** as the *cause*.

Translation

In some environments, the external representation of a text file is different from its internal representation: for example, in MS-DOS, text files on disk contain CR-LF, and in memory contain only LF at the end of each line. Binary streams (**BinIO.instream**) match the external files byte for byte; text streams (**TextIO.instream**) are translated. Normally, users of **TextIO** will not need to know or care about this translation.

The functions **IO.translateRd** (and **IO.translateWr**) perform the "identity" translation on readers (and writers); that is, each BinIO byte is translated to a TextIO char using **Char.chr**. Each ML implementation may also make nontrivial translation functions available to the user as part of the operating-system-dependent modules. For example, there might be a MicrosoftWindows structure containing a substructure with a **translateRd** function to delete carriage returns, and a **translateWr** function that would insert them. See also section 9.4.

4 PRIM_IO

Primitive I/O is meant to be an abstraction of the system call operations commonly available on file descriptors.

For basic "operating system" functions such as reading and writing, the input/output modules do not reference the **OS** structure directly. Instead, each stream is built on a **PrimIO.reader** or **PrimIO.writer**; the readers and writers contain functions that can accomplish the system calls. But it is also possible for users to synthesize readers or writers that don't do system calls at all, or do unconventional ones.

```
signature PRIM_IO =
sig
    type elem
    type vector
    type array
    type pos
    val posLess : pos * pos -> bool
    datatype reader = Rd of
              {readBlock :   (int -> vector) option,
               readaBlock:   ({data: array, first: int, nelems: int} ->
                               int) option,
               readNoBlock : (int -> vector option) option,
               readaNoBlock: ({data: array, first: int, nelems: int} ->
                               int option) option,
               block     : (unit -> unit) option,
               canInput  : (unit -> bool) option,
               name      : string,
               chunkSize : int,
               close     : unit -> unit,
               getPos    : unit -> pos,
           findPos   : {data: vector, first: int, nelems: int}*pos -> pos,
               setPos    : (pos -> unit),
               endPos    : (unit -> pos),
           iodesc    : OS.IO.io_desc option
              }
    datatype writer = Wr of
               {writeNoBlock: ({data: vector, first: int, nelems: int} ->
                               int option) option,
               writeaNoBlock: ({data: array, first: int, nelems: int} ->
                                int option) option,
               writeBlock: ({data: vector, first: int, nelems: int} ->
                             int) option,
               writeaBlock: ({data: array, first: int, nelems: int} ->
                              int) option,
               block: (unit->unit) option,
               canOutput: (unit->bool) option,
               name: string,
               chunkSize: int,
           lineBuf: (elem -> bool) option,
               close: unit -> unit,
               getPos : (unit->pos),
               setPos : (pos->unit),
               endPos : (unit->pos),
           iodesc    : OS.IO.io_desc option}
    val augmentIn : reader -> reader
    val augmentOut: writer -> writer
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 **reader** are

close()
: Closes the reader (for example, frees operating system resources). Further operations (other than **close**) to this reader are illegal and must be checked for by the reader (the **IO.ClosedStream** exception must be raised). However, a **close** operation on a closed reader will not raise an exception.

name
: The name associated with this file or device, for use in error messages shown to the user. For special streams such as *stdIn*, the name field is implementation-dependent.

chunkSize
: The recommended (efficient) size of read operations on this reader. This is typically the block size of the operating system's buffers, or a size that the ML runtime system can handle efficiently. 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.

readNoBlock(n)

(optional) Reads i elements without blocking, for $1 < i \leq n$, creating a vector v, returning SOME(v); or (if end of file is detected without blocking), returns SOME(*empty*); or (if a read would block) returns NONE.

readBlock(n)

(optional) Reads i elements for $1 \leq i \leq n$ returning a vector v of length i; or (if end of file is detected) returns an empty vector. Blocks (waits) if necessary until end of file is detected or at least one element is available. To achieve "block until exactly n elements have been read" it is necessary to loop on **readBlock**, because each call only guarantees to block until at least one element is ready.

readaNoBlock{buf=a,first=i,nelems=n}

(optional) Reads k elements without blocking, for $1 \leq k \leq n$ 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.

readaBlock{buf=a,first=i,nelems=n}

(optional) Reads k elements for $1 \leq k \leq n$ 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 reamain before end-of-file, returns 0.

block()

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

canInput()

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

$p =$ **getPos**()

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

The **getPos** function must be nondecreasing (in the absence of **setPos** operations, or other interference to the underlying object). Where **setPos** is not provided, the reader can just count the elements returned from read operations and **getPos** can tell the count. But an implementation of **getPos** that always returns zero is legal.

$p' =$ **findPos**($\{\text{data} = v, \text{first} = i, \text{nelems} = n\}, p$)

Tells the position p' of the $(i+n)$th element of the vector v, assuming that the position of the ith element is p. Section 9.4 explains why this is useful.

setPos(i)

(optional) Move to position i in file. Optional, in the sense that it may raise an exception if unimplemented or invalid.

endPos()

The position at the end of the file. Optional, in the sense that it may raise an exception if unimplemented, or invalid on this reader.

iodesc()

The I/O descriptor (an abstraction of operating-system file descriptor) associated with this stream, if any.

Providing more of the optional functions increases functionality and/or efficiency of clients:

1. Absence of all of **readBlock**, **readaBlock**, and **block** means that blocking input is not possible.
2. Absence of all of **readNoBlock**, **readaNoBlock**, and **canInput** means that non-blocking input is not possible.

3. Absence of **readNoBlock** means that non-blocking input requires two system calls (using **canInput, readBlock**).

4. Absence of **readaNoBlock** or **readaBlock** means that input into an array requires extra copying. *But the "lazy functional stream" model does not use arrays at all.*

   The **augmentIn** function takes a reader r and produces a reader in which as many as possible of **readBlock**, **readaBlock**, **readNoBlock**, **readaNoBlock** are provided, by synthesizing these from the operations of r. For example, **augmentIn** can synthesize **readBlock** from **readNoBlock+block**, synthesize vector reads from array reads, synthesize array reads from vector reads, as needed.

   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. Providing more than the minimum by just doing the obvious synthesis inside the PrimIO layer is not recommended because then clients won't get the "hint" about which are the efficient ("recommended") operations.

5. Absence of **endPos** means that very large inputs (where vectors must be pre-allocated) cannot be done efficiently (in one system call, without copying).

6. The client is likely to call **getPos** on every read operation. Thus, the reader should maintain its own count of (untranslated) elements to avoid repeated system calls. This should not be done on streams opened for atomic append, of course, where the information cannot be obtained except by a system call.

7. Absence of **setPos** prevents random access.

8. The **findPos** function is needed in conjunction with readers that do translation, so that positions do not always correspond 1–1 to elements returned from **read**. If the translation function is invertible, then **findPos** will be straightforward to implement. If not invertible, then **findPos** can seek to *pos* in the underlying file, and re-translate forward to the right point. In that case, the implementation of **findPos** will probably require: $p_0 = $ **get**(), **setPos**(*pos*), **read**, **setPos**(p_0) to restore the file position to what it was before the **find** operation.

9. Readers that do no translation, so that positions do correspond 1–1 to elements returned from the **read** functions, can provide a very simple **findPos** function:

```
fun find({data,first,nelems},p) =
      Position.+(p, Position.fromDefault nelems)
```

10. Readers whose **getPos** always returns zero should also have a **findPos** that always returns zero.

The components of a **writer** are:

writeBlock{buf=v,first=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.

writeaBlock{buf=a,first=i,nelems=n}
This (optional) function writes elements $a_i, \ldots, a_{i+k-1}$, 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.

writeNoBlock{buf=v,first=i,nelems=n}
This (optional) function writes elements $v_i, \ldots, v_{i+k-1}$, for $0 < k \leq n$ to the output device without blocking, and returns SOME(k); or (if the write would block) returns NONE.

writeaNoBlock{buf=a,first=i,nelems=n}
This (optional) function writes elements $v_i, \ldots, v_{i+k-1}$, for $0 < k \leq n$ to the output device without blocking, and returns SOME(k); or (if the write would block) returns NONE.

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

canOutput()
(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. For special streams such as *stdOut*, the name field is implementation-dependent.

chunkSize
The recommended (efficient) size of write operations on this writer. This is typically the block size of the operating system's buffers, or a size that the ML runtime system can handle efficiently. 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 (functions in other modules taking writers as arguments may raise exceptions).

lineBuf
On "line-buffered" streams, output should be flushed to the underlying reader whenever a newline is written. When **lineBuf**=SOME(f), then the writer is line-buffered, and the buffer should be flushed on any element x such that $f(x) = true$. This does not affect the semantics of other writer operations, and is only a recommendation to the higher-level (StreamIO) buffering layers.

close()
Closes the writer (for example, frees operating system resources devoted to this writer). Further operations (other than **close**) to this writer are illegal (will raise ClosedStream) and must be checked for by the writer.

getPos()
Tells the current position within the file. Most useful on seekable writers. Optional, in the sense that it may raise an exception if unimplemented or invalid.

endPos()
The position at the end of the file. Optional, in the sense that it may raise an exception if unimplemented or invalid.

setPos(i)
Moves to position i in the file, so future writes occur at this position. Optional, in the sense that it may raise an exception if unimplemented or invalid.

iodesc()
The I/O descriptor (an abstraction of operating-system file descriptor) associated with this stream, if any.

One of **writeBlock**, **writeaBlock**, **writeNoBlock**, or **writeaNoBlock** must be provided. Providing more of the optional functions increases functionality and/or efficiency of clients:

1. Absence of all of **writeBlock**, **writeaBlock**, and **block** means that blocking output is not possible.

2. Absence of all of **writeNoBlock**, **writeaNoBlock**, and **canOutput** means that non-blocking output is not possible.

3. Absence of **writeNoBlock** means that non-blocking output requires two system calls (using **canOutput**, **writeBlock**).

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

5. Absence of **writeaNoBlock**, **writeNoBlock**, and **canOutput** from a writer means that nonblocking output is impossible. But the standard **StreamIO** modules do not support nonblocking output anyway.

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

Unlike readers, which can expect their **getPos** functions to be called frequently, writers need need not implement **getPos** in a super-efficient manner: a system call for each **getPos** is acceptable. Furthermore, **getPos** need not be supported for writers (it may raise an exception), whereas for readers it must be implemented (even if inaccurately).

The **augmentOut** function takes a writer w and produces a writer in which as many as possible of **writeBlock**, **writeaBlock**, **writeNoBlock**, **writeaNoBlock** are provided, by synthesizing these from the operations of w.

Exceptions The PrimIO functions (component fields of readers and writers) may raise the following exceptions:

Subscript for any function taking the {*data, first, nelems* } type, if *first* and *nelems* imply an out-of-bounds reference to *data*.

SysErr for any function that performs a system call.

ClosedStream for attempted operations on closed readers or writers.

- Other exceptions as needed for special purposes (unconventional readers and writers).

Readers and writers should not, in general, raise the **Io** exception.

5 PrimIO

The functor **PrimIO** builds standard instances of the **PRIM_IO** signature.

```
  functor PrimIO(structure A : MONO_ARRAY
                 structure V : MONO_VECTOR
                 sharing type A.elem=V.elem
                 sharing type A.vector=V.vector
                 val someElem : A.elem
                 type pos) : PRIM_IO =
   struct . . .  end
```

The only nontrivial parts of the PrimIO functor are the implementations of the functions **augmentIn**, and **augmentOut**, which simulate one kind of reader (or writer) functionality in terms of other kinds. For example:

```
fun augmentIn (r as Rd r') =
   let fun readaToRead reada i =
             let val a = A.array(i,someElem)
                 val i' = reada{data=a,first=0,nelems=i};
              in A.extract(a,0,i')
             end
       fun stripOption (SOME x) = x

       val readBlock' =
          case r
           of Rd{readBlock=SOME f,...} => SOME f
            | Rd{readaBlock=SOME f,...} => SOME(readaToRead f)
            | Rd{readNoBlock=SOME f,block=SOME b,...} =>
```

```
                 SOME(fn i => (b(); stripOption(f i)))
        | Rd{readaNoBlock=SOME f, block=SOME b,...} =>
                 SOME(fn i => (b(); stripOption(readaToRead f i)))
        | _ => NONE
  . . .
 in Rd{block= #block r', . . . readBlock=readBlock', . . . }
end
```

6 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 *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 subvector
    type pos    sharing type pos = PrimIO.pos
    type instream
    type outstream
    val mkInstream   : PrimIO.reader -> instream
    val closeIn      : instream -> unit
    val setPosIn     : instream * pos -> instream
    val getPosIn     : instream -> pos
    val endPosIn     : instream -> pos
    val input         : instream -> vector * instream
    val inputAll     : instream -> vector
    val inputNoBlock : instream -> (vector * instream) option
    val input1        : instream -> elem option * instream
    val inputN       : instream * int -> vector * instream
    val endOfStream : instream -> bool
    val getReader    : instream -> PrimIO.reader
    val mkOutstream : PrimIO.writer -> outstream
    val closeOut : outstream -> unit
    val output    : (outstream * vector) -> unit
    val outputS   : (outstream * subvector) -> unit
    val output1   : (outstream * elem) -> unit
    val flushOut : outstream -> unit
    val getPosOut  : outstream -> pos
    val setPosOut  : outstream * pos -> unit
    val endPosOut  : outstream -> pos
    val getWriter: 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 $(v, f') =$ **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.

subvector
For text I/O, **subvector** will be the **substring** type. For binary I/O, there is no notion of substring, so **subvector** is the same as **vector** and is not very useful.

$f =$ **mkInstream**(r)
Create a buffered stream f from a reader r. (Most users will normally use **TextIO.openIn** instead.)

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

$g =$ **setPosIn**(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.*

getPosIn(f)
Return the current position of f. *Not always supported.*

endPosIn(f)
Return the position at end of file of f. *Not always supported.*

$(v, f') =$ **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.

$v =$ **inputAll**(f)
Return the vector v of all the elements of f up to end of stream. Semantically equivalent to:

```
fun inputAll(f) = let val (a,f') = input f
                   in if size(a)=0 then a
                      else a ^ inputAll f'
                  end
```

where ^ is the concatenation operator on element vectors.

$(v, f') =$ **inputNoBlock**(f)
If any non-empty sequence v of f is available or can be read from the operating system without blocking, return SOME(w, f') where w is any non-empty prefix of v, and f' is the "rest" of the stream. Otherwise return NONE. On streams that do not support non-blocking input, raise General.NonblockingNotSupported.

$(c, f') =$ **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 NONE. Otherwise read from the operating system (which may block) until one of those conditions occurs. Semantically equivalent to:

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

$(v, f') =$ **inputN**(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.*) Semantically equivalent to:

```
fun inputN(f,0) = (empty, f)
  | inputN(f,n) = case input1 f
                   of (NONE, f') => (empty, f')
                    | (SOME x, f') =>
                       let val (s,f'') = inputN(f,n-1)
                            in (x^s, f'')
                       end
```

endOfStream(f)

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

getReader(f)

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

```
let val r = getReader f
    val v = inputAll f
 in ...
end
```

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

$f =$ **mkOutstream**(w, s)

Create a buffered outstream f from a writer w. In w, **writeBlock**, **writeaBlock**, and **block** must not all be NONE or an **Io** exception will be raised.

closeOut(f)

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

flushOut(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; this may block until the system is prepared to accept more output. **StreamIO** does not provide any nonblocking output function.

outputS(f, v)

Write the subsequence (substring) v to f.

output1(f, x)
  Write the element x to f; may block.

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

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

endPosOut(f)
  The position at the end of file f. *Not always supported.*

setPosOut(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**.

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 _ = closeIn f
    val (b,_) = input f
 in  a=b andalso endOfStream f'
end (* must be true *)
```

Closing a terminated stream is legal and harmless:

```
  (closeIn f; closeIn f; true)
```

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

```
let val a = input f
 in case inputNoBlock f
     of SOME a => a=b
      | NONE => false
end (* must be true *)
```

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

A freshly opened stream is still undetermined (no "read" has yet been done on the underlying reader):

```
let val a = TextIO.openIn name (* or  mkInstream(r),
                                  or  BinIO.openIn name  *)
 in close a;
    size(input a) = 0
end
```

This has the useful consequence that if one opens a stream, then extracts the underlying reader, the reader has not yet been advanced in its file.

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

```
(case (close f; inputNoBlock f) of SOME _ => true | NONE => false)
```

The **endOfStream** test is equivalent to **input** returning an empty sequence:

```
let val (a,_) = input f   in  (size(a)=0) = (endOfStream f)   end
```

getPosIn is accurate even if two different instreams are created from the same reader and they interleave operations. Thus, the implementation of StreamIO must make no assumption that the position at the end of one **read** operation is the same as the position at the beginning of the next.

Unbuffered I/O That is, if chunkSize=1 in the underlying reader, then **input** operations must be unbuffered:

```
 let val f = mkInstream(reader)
     val (a,f') = input(f,n)
     val PrimIO.Rd{chunkSize,...}=getInstream f
   in chunkSize>1 orelse endOfStream f'
 end
```

Though **input** may perform a **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:

```
 let val f = mkInstream(reader)
     val _ = doInputOperationsOn(f)
     val (a,f') = input(f,n)
     val PrimIO.Rd{chunkSize,...}=getInstream f
   in chunkSize>1 orelse endOfStream f'  (* could be false*)
 end
```

because in this case, the stream *f* may have accumulated a history of several responses, and **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 **output** returns.

Don't bother the reader **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 **endOfFile** will not make repeated system calls.

This rule could be formalized by defining a "monitor:"

```
val monitor: reader -> {rd: reader,
                         charsRead: int ref,
                         opCount: int ref}
```

and making statements such as:

```
let val {rd,charsRead,opCount} = monitor(reader)
    val f = mkInstream(rd)
    val (f',nElems) = doThingsCountingElements(f)
    val p1 = getPosIn f'
    val c1 = !charsRead
    val ops = !opCount
    val _ = input f'
 in not ((nElems < c1) andalso (!opCount > 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 cntl-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:

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

The "don't bother the reader" rule, combined with the definition of **endOfStream**, guarantees that

```
endOfStream(f) = endOfStream(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.

Line Buffering When a string containing a newline character is written to a line-buffered outstream, the buffer (up to and including the newline, at least) should be written to the underlying writer. PrimIO writers with **lineBuf** = SOME(f) are converted to line-buffered outstreams by **mkOutstream**.

When input is demanded from a line-buffered instream, and this input requires a read operation on the underlying reader, all line-buffered writers must be flushed. This is done by creating a line-buffered reader and applying **mkInstream** to it. Line-buffered readers look just like other readers from the outside, but their **readBlock** (etc.) functions perform buffer-flushing on some list of outstreams. Creation of line-buffered readers is the job of the operating-system-dependent IO interface module (as is the creation of many other readers). There is no special handling of line-buffered readers or instreams within the StreamIO implementation, as there must be for line-buffered writers.

7 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 someElem : PrimIO.elem
                 val posLess : PrimIO.pos * PrimIO.pos -> bool
                 val posDiff : ({lo: PrimIO.pos, hi: PrimIO.pos} -> int) option
                 type subvector
                 val base: subvector -> Vec.vector * int * int
               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 **someElem** is used to initialize buffer arrays; any element will do.

The types **instream** and **outstream** in the result of the **StreamIO** functor must be abstract.

If **flushOut** finds that it can do only a partial write (i.e., **writeaBlock** or a similar function returns a "number of elements written" less than its "nelems" argument) then **flushOut** must adjust its buffer for the items written and then try again. If the first or any successive write attempt returns zero elements written (or raises an exception) then **flushOut** raises an **Io** exception.

If an exception occurs during any **StreamIO** operation, then **StreamIO** must, of course, leave itself in a consistent state, without losing or duplicating data.

In some ML systems, a user interrupt aborts execution and returns control to a top-level prompt, without raising any exception that the current execution can handle. It may be the case that some information must be lost or duplicated. Data (input or output) must *never* be duplicated, but may be lost. This can be accomplished without **StreamIO** doing any explicit masking of interrupts or locking. On output, the internal state (saying how much has been written should be updated *before* doing the *write* operation; on input, the *read* should be done before updating the count of valid characters in the buffer.

StreamIO does not need **PrimIO.pos** to be any kind of integer, but it must be a total ordering with a total and irreflexive comparison operator **posLess** supplied, and the **PrimIO** read operations must semi-monotonically increase the position values.

Subvectors are a generalization of substrings. When **StreamIO** is used to implement **TextIO**, one can use

```
type subvector = Substring.substring
val base = Substring.base
```

For binary (or other) I/O, where it is not necessary that **outputS** do anything useful, one can provide a "dummy" implementation of `subvector`; for example:

```
type subvector = Vec.vector
fun base v = (v,0,Vec.length v)
```

Implementation notes:

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

With buffered reading, a **getPosIn** operation on the **instream** may be done in the middle of a buffer. Calculating this requires knowing the position of the beginning of the buffer, and using **findPos**. But this means that the **StreamIO** system must do a **getPos** just before reading each new buffer, and remember that position.

Here are some suggestions for efficient performance:

- Operations on the underlying readers and writers (**readBlock**, etc.) are expected to be expensive (involving a system call, with context switch).
- Small input operations can be done from a buffer; the **readBlock** or **readNoBlock** operation of the underlying reader can replenish the buffer when necessary.
- Each reader may provide only a subset of **readBlock**, **readNoBlock**, **block**, **canInput**, etc. An augmented reader that provides more operations that can be constructed using **PrimIO.augmentIn**; but it may be more efficient to use the functions directly provided by the reader, instead of relying on the constructed ones. The same applies to augmented writers.
- 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 **inputAll** or **inputN** 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, **readBlock**(n). However, in a typical implementation of **readBlock** this requires pre-allocating a vector of size n. If the user does **inputAll**() or **inputN**(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 **endPos**, subtract the current position, and try to **read** that much. But one should also keep things rounded to the nearest **chunkSize**.

- Subtracting the current position is difficult (!) if **pos** is an abstract type. The optional function **posDiff** is provided to compute (even approximately) the distance (in elements) between two positions. A slight overestimate in the computation is better than a slight underestimate.
- The use of **endPos** to try to do (large) read operations of just the right size will be inaccurate on translated readers. But this inaccuracy can be tolerated: if the translation is anything close to 1–1, **endPos** will still provide a very good hint about the order-of-magnitude size of the file.
- Similar suggestions apply to very large **output** operations. Small outputs go through a buffer; the buffer is written with **writeaBlock**. Very large outputs can be written directly from the argument string using **writeBlock**.
- A lazy functional 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 **readNoBlock** if it exists, otherwise do **canInput** followed (if appropriate) by **readBlock**.
- To support blocking input, use **readBlock** if it exists, otherwise do **readNoBlock** followed (if would block) by **block** and then another **readNoBlock**.
- To support lazy functional streams, **readaBlock** and **readaNoBlock** are not useful; they are included only for completeness.
- **SetPosIn**, if setPos-ing forward, might choose to follow the buffer sequence, and can perhaps satisfy the **setPos** request without any underlying reader operation.
- **GetPosIn**, 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.
- **writeaBlock** should, if necessary, be synthesized from **writeBlock**, and vice versa. Similarly for **writeaNoBlock** and **writeNoBlock**; **readaNoBlock** and **readNoBlock**; **readaBlock** and **readBlock**.

8 ImperativeIO functor

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

```
functor ImperativeIO(structure S : STREAM_IO) : IMPERATIVE\_IO = ...
```

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

The semantics of **ImperativeIO** are simple enough that it is sufficient to give a reference implementation, without much explanation.

```
functor ImperativeIO(structure S : STREAM_IO) : IMPERATIVE_IO =
let abstraction I : sig include IMPERATIVE_IO sharing StreamIO=S end =
 struct
   structure StreamIO = S
   type instream = S.instream ref
   type outstream = S.outstream ref
   type elem = S.elem
```

```
        type vector = S.vector
        type subvector = S.subvector
        type pos = S.pos
        val mkInstream = ref
        val getInstream = !
        val setInstream = op :=
        val mkOutstream = ref
        val getOutstream = !
        val setOutstream = op :=
        fun endOf f = if S.endOfStream f then f else endOf(#2(S.input f))
        fun closeIn(r as ref f) = (S.closeIn f; r := endOf f)
        fun setPosIn(r as ref f, i) = r := S.setPosIn(f,i)
        val getPosIn = S.getPosIn o !
        val endPosIn = S.endPosIn o !
        fun inputN (r as ref f, n) = let val (v,f') = S.inputN(f,n) in r:=f'; v end
        fun input (r as ref f) = let val (v,f') = S.input f in r:=f'; v end
        fun input1 (r as ref f) = let val (v,f') = S.input1 f in r:=f'; v end
        fun inputNoBlock (r as ref f) =
            case S.inputNoBlock f
             of SOME(v,f') => (r := f'; SOME v)
              | NONE => NONE
        fun inputAll(r as ref f) = let val v = S.inputAll f
                                    in r := endOf f; v end
        val endOfStream = S.endOfStream o !
        fun lookahead(ref f) = #1(S.input1 f)
        val closeOut = S.closeOut o !
        fun output(ref f, v) = S.output(f,v)
        fun outputS(ref f, v) = S.outputS(f,v)
        fun output1(ref f, x) = S.output1(f,x)
        val getPosOut = S.getPosOut o !
        fun setPosOut(ref f, i) = S.setPosOut(f,i)
        val endPosOut = S.endPosOut o !
        val flushOut = S.flushOut o !
       end
      in I
     end
```

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

```
fun endOfStream (f as ref ff) = StreamIO.endOfStream ff
```

This implies

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

Furthermore, second call to **endOfStream** 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 STREAM_IO and IMPERATIVE_IO. The only purpose of the extra **ref** in IMPERATIVE_IO is to allow "output redirection."

9 Application Notes

9.1 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.getReader
                    (TextIO.getInstream(TextIO.openIn name))
val writer = TextIO.StreamIO.getWriter
                    (TextIO.getOutstream(TextIO.openOut name))
```

2. Do some buffered writes; then discard the buffering layer.

```
let val out = TextIO.mkOutstream(TextIO.StreamIO.mkOutstream(writer))
 in TextIO.setPosOut(out,somePos);
    output(out,"Hello ");
    output(out,"World\n");
    flushOut out
end
```

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

```
let val inf = TextIO.mkInstream(TextIO.StreamIO.mkInstream(reader))
 in TextIO.setPosIn(inf,anotherPos);
    input inf;
    input inf
end
```

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

9.2 Other reader/writer devices

The functions **TextIO.openIn** and **TextIO.openOut** provide system-default ways to create streams (whose underlying readers and writers can be extracted), from "file names."

SML implementations are likely to provide other ways to create readers and writers. For example,

```
structure Socket :
   sig  type socketName
        structure P = TextIO.StreamIO.PrimIO
        val openSocketTextReader: socketName -> P.reader
        val openBidirectionalSocket: socketName ->
                    P.reader * P.writer
        . . .
   end
```

Then the user could buffer these readers by using **mkInstream**.

Alternatively, a **Socket** interface could provide the high-level **instream**:

```
structure Socket :
   sig  type socketName
        val openSocketTextIn: socketName -> TextIO.instream
        val openBidirectionalSocket: socketName ->
                    TextIO.instream * TextIO.outstream
        . . .
    end
```

and the user could extract the reader by using **getInstream** and **getReader**.

9.3 String readers/writers

A useful kind of reader/writer is an internal text queue, not using any devices at all:

```
local
 fun primPipe() : TextIO.StreamIO.PrimIO.reader *
                     TextIO.StreamIO.PrimIO.writer =
            . . .
 in
    fun pipe() : instream * outstream =
              (* layer mkInstream and mkOutstream on
                 components of primPipe() *)
end
```

It would be natural to provide such functions in a library.
Here's an even simpler example:

```
 fun stringReader(source : string) : TextIO.StreamIO.PrimIO.reader =
  let val pos = ref 0
      fun read n = let val p = !pos
                       val m = min(n, size source - p)
                    in pos := p+m; substring(source,p,m)
                   end
  Rd{readNoBlock = SOME(fn n => SOME(read n)),
     readaNoBlock = NONE,
     readBlock = SOME(read),
     readaBlock= NONE,
     block = SOME(fn()=>()),
     canInput = SOME(fn()=>true),
     name="<string>",
     chunkSize=size source,
     close=fn()=>(),
     getPos=fn()=>Position.fromDefault(!pos),
     findPos=fn({data,first,nelems},p)=>p+nelems,
     setPos=((fn k => if 0<=k andalso k <= size source then pos:=k
                   else raise Fail "position out of bounds")
                   o Position.toDefault),
     endPos=(fn()=>Position.fromDefault(size source)),
     iodesc=NONE}
  end

 val openString : string -> instream =
       TextIO.mkInstream o TextIO.StreamIO.mkInstream o stringReader
```

9.4 Translated readers

Sometimes one wants to apply a translation function to a stream. For example, one might want to translate CR-LF to LF on input, or translated escape-coded ASCII into Unicode. I shall discuss translated input streams (readers) here, but the same ideas apply to translated output streams (writers).

Since anyone is allowed to counterfeit a reader, it is easy to write a translation function on readers:

```
fun translate1 (source: TextIO.PrimIO.reader) : TextIO.PrimIO.reader
 or
fun translate2 (source:  BinIO.PrimIO.reader) : TextIO.PrimIO.reader
```

Here's an example:

```
 fun remove_CR(rd0 as TextIO.StreamIO.PrimIO.Rd rd) :
                           TextIO.StreamIO.PrimIO.reader =
  let fun charCR(#"\013") = ""
        | charCR c = implode c
      fun stringCR s = concat(mapChar charCR (s,0,size s))
      fun option f NONE = NONE
        | option f (SOME x) = SOME(f x)
      fun retranslate(_,0,pos) = pos
        | retranslate(read,nelems,pos) =
          let val s = read nelems
              val len = size s
              fun loop(i,n,p) = if i=s then retranslate(read,n,p)
                                else if n=0 then p
                                else if CharVector.sub(s,i)= #"\013"
                                         then loop(i+1,n,p)
                                else loop(i+1,n-1,p)
           in loop(0,nelems,pos)
          end
   in TextIO.StreamIO.PrimIO.Rd{
     readNoBlock = option (fn get =>  option stringCR o get)
                          (#readNoBlock rd,)
     readaNoBlock = (* etc. *),
     readBlock = option (fn get => stringCR o get) (#readBlock rd),
     readaBlock=  (* etc. *),
     block = #block rd,
     canInput = #canInput rd,
     name= #name rd,
     chunkSize= #chunkSize rd,
     close= #close rd,
     setPos=#setPos rd,
     endPos=#endPos rd,
     getPos=#getPos rd,
     findPos=
       case (TextIO.StreamIO.PrimIO.augmentIn rd0)
        of TextIO.StreamIO.PrimIO.Rd{readBlock=SOME readb,...} =>
           fn ({data,first,nelems},pos)=>
                            let val p0 = #getPos rd ()
                               val p1 = #setPos rd pos;
                                         retranslate(readb,nelems,pos))
                            in #setPos rd p0; p1
                           end}
            | _ => raise Fail "Cannot findPos",
     iodesc=NONE}
  end
```

Note that the positions in this translated reader (and thus in the translated stream) do not correspond 1–1 to positions in the underlying reader. Thus, **findPos** must be implemented. A good, simple solution is to avoid random access on translated streams:

```
 findPos = fn _ => raise Fail "Cannot findPos"
```

But here we have chosen to provide **findPos** whenever possible. Because the translation is not invertible (we don't know where the CR characters might have been), **findPos** must re-read the original stream.

Users who need to do random access on translated streams might alse use a solution similar to the one in section 9.1: do **setPos** on the underlying, untranslated reader. Then, after each **setPos**, apply afresh the translation function (such as **remove_CR** and then apply a new buffer (via **mkInstream**).

9.5 Abstract positions

In applications where one wants seekable, translated readers with "moded escapes" it is difficult represent positions as integers. This will happen if escape characters semi-permanently change the translation state of a stream, rather than affecting just the next character.

In such a case, one might want to have an abstract data type *position*, with a total ordering but without a mapping to integers.

One way to accomplish this is to make a new structure matching the **PRIM_IO** signature:

```
abstraction MyTextPrimIO : PRIM_IO =
    PrimIO(structure A = CharArray
           structure V = CharVector
           val someElem = #"\000"
           type pos = MyPosType.pos)
```

Now one can write translated readers that can deal with translated positions more flexibly.

The **StreamIO** functor can be used to create a buffered I/O system for these new readers/writers:

```
structure MyStreamIO =
     StreamIO(structure PrimIO = MyPrimIO
              . . .
              val posLess = MyPosType.<
              val posDiff = NONE)
```

Or, `posDiff=SOME(MyPosType.-)` if possible.

Now MyStreamIO.instream is a different type than TextIO.StreamIO.instream, so one cannot use the same function to operate on both kinds of instreams unless this function is in a functor parameterized by **StreamIO**.

Also, it is possible to write a function to translate a **MyPrimIO.reader** into an ordinary **PrimIO.reader** (but with **setPos** disabled):

```
 val NoRandomAccess = Fail "Random access not supported on this stream"

 fun standardize (MyPrimIO.Rd rd) =
     TextIO.StreamIO.PrimIO.Rd{
         readNoBlock = #readNoBlock rd,
         readaNoBlock = #readaNoBlock rd,
         readBlock = #readBlock rd,
         readaBlock=  #readaBlock rd,
         block = #block rd,
         canInput = #canInput rd,
         name= #name rd,
         chunkSize= #chunkSize rd,
         close= #close rd,
         getPos=fn _ => 0,
         findPos=fn _ => raise NoRandomAccess,
         setPos=fn _ => raise NoRandomAccess,
         endPos=fn _ => raise NoRandomAccess,
         iodesc=#iodesc rd}
```

9.6 Lexical analysis

Lexical analyzers need to process their input efficiently, and often need some amount of lookahead. Line-oriented applications need to read one line of text at a time, efficiently. Both of these applications can make effective use of lazy-stream input.

Consider the implementation of an **inputLine** function, that reads up to the next newline character. A naive implementation would read characters, then concatenate them:

```
fun inputLine (f: TextIO.instream) =
 let fun loop () = case input1 f
                   of SOME(#"\n") => nil
                    | SOME c => c :: loop()
                    | NONE => nil
  in implode (loop())
 end
```

Now, we may wish to avoid all the list construction and **implode** call. Thus:

```
fun inputLine (f: TextIO.instream) =
 let val g0 = TextIO.getInstream f
     fun loop(i,g) = case input1 g
                   of (SOME(#"\n"),_) =>
                         let val s = TextIO.inputN(f,i)
                          in TextIO.input1 f; s
                         end
                    | (SOME c, g') => loop(i+1,g')
                    | (NONE,_) => TextIO.inputN(f,i)
  in loop(0,g0)
 end
```

This has the effect of looking through the input buffer for a newline character, then extracting just the right-length string from the input buffer; but it's all done abstractly.

There are no list constructions, and only one string copy: the extract implied by the **inputN** call. On the other hand, there **is** a function call for each character; I do not see this as a problem. We expect ML programs (or, in fact programs in any language) to implement abstract data types via a function-call interface; if this becomes a source of inefficiency, perhaps the solution is for compilers to implement cheaper function calls.

A very similar approach works for lexical analyzers which do more general (perhaps multi-character) lookahead: First scan the lazy stream to determine the length of the token, then use **inputN** to extract it and advance the stream.

InputLine is provided as a standard function of **TextIO**, but this implementation is explained to illustrate how variations on it can be constructed.