Proposed interface for Standard ML Stream I/O

Andrew W. Appel

November 30, 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.[1][2]

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.[3]

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

      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 * pos -> unit
      val getpos_in      : instream -> pos
      val endpos_in      : instream -> pos

      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 -> pos
```

[1]All footnotes in this document indicate unresolved issues. Issues should be resolved, and footnotes removed, by the last draft.

[2]The conventions for multi-word identifiers (underscores vs. capitalization) in the initial basis are still unresolved. Therefore, I have been inconsistent in my punctuation.

[3]Perhaps it would be better to separate input and output in the the IO modules and signatures. There seems to be no reason not to do this, except that it means twice as many modules.

```
      val endpos_out  : outstream -> pos
      val setpos_out  : outstream * pos -> unit

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

      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
  end

structure FilePosInt : INTEGER

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

structure BinIO : BIN_IO

signature TEXT_IO =
  sig
     include IO
     sharing type StreamIO.elem = char
     sharing type StreamIO.vector = string
     sharing type pos=FilePosInt.int
     val std_in : instream
     val std_out: outstream
     val std_err: outstream
     val translate_in: BinIO.StreamIO.PrimIO.reader ->
                       TextIO.StreamIO.PrimIO.reader
     val translate_out: BinIO.StreamIO.PrimIO.writer ->
                       TextIO.StreamIO.PrimIO.writer
  end
```

```
structure TextIO : TEXT_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**).

$f =$ **open_in**(s)

Opens a file named s as a stream f.[4]

close_in(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).

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

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

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.

$c =$ **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.

$v =$ **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

[4]The **BinIO** and **TextIO** modules, as well as all their substructures, provide general operations on streams, no matter how these streams are opened. Thus, they are almost entirely operating-system independent. Putting **open_in** and **open_out** in these modules introduces an operating-system dependency. It is intended that users or vendors may provide some structure **X** matching the **IO** signature; such a structure may not want to provide any **open_in** of this form (taking a string argument). In that case, **X.open_in** should raise an exception. This is slightly inelegant; Berry and Reppy would like to see **open_in** removed from the **IO** signature and put in an operating-system dependent structure such as **OS.File**. I put the functions in **IO** for users' convenience, but I could be convinced to remove them.

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.)

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)`.

$c =$ **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 6.

setpos_in(f, i)
Seek to position i in f. *Not always supported (raises* **Io** *if not supported on* f*).*

$i =$ **getpos_in**(f)
Tell the current position of f. For the standard modules **TextIO** and **BinIO**, i is an integer equal to the number of elements since the beginning of the file. Positions correspond 1–1 to elements in the file, and are not in any way abstract. *Not always supported (raises* **Io** *if not supported on* f*).*

$i =$ **endpos_in**(f)
Tell the ending position of f. For the standard modules **TextIO** and **BinIO**, i is an integer equal to the number of elements in the file. *Not always supported (raises* **Io** *if not supported on* f*).*

Operations on outstreams

$f =$ **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.

$i =$ **getpos_out**(f)

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

$i =$ **endpos_out**(f)

Tell the ending position of f *(not always supported).* For the standard modules **TextIO** and **BinIO**, i is an integer equal to the number of elements in the file. *Not always supported (raises* **Io** *if not supported on* f*).*

setpos_out(f, i)

Seek to position i of f *(not always supported).* For the standard modules **TextIO** and **BinIO**, i is an integer equal to the number of elements since the beginning of the file. Positions correspond 1–1 to elements in the file, and are not in any way abstract.

Any of these functions may raise the **Io** exception if an operation fails (including **close_out** if a buffer cannot be flushed).

Random access

The **getpos, endpos, setpos** functions all operate on special **FilePosInt** integers. In some implementations these may be ordinary integers, in others they may be double-precision or even arbitrary precision integers. Thus, one of the following is likely to be true:

```
    structure FilePosInt = Int
or  structure FilePosInt = LargeInt
```

This allows operations on extremely large files.

Users can operate on the **pos** type using **FilePosInt.+** and **FilePosInt.-**; or convert to/from ordinary integers using **FilePosInt.toDefault** and **FilePosInt.fromDefault**—at the risk of being unable to process large files.[5]

Closing files on program exit

All streams created by **TextIO.open_in**, **TextIO.open_out**, **BinIO.open_in**, and **BinIO.open_out** will be closed (the outstreams among them flushed) when the ML program exits. The outstreams **std_out** and **std_err** will be flushed, but not closed, on program exit.

[5]Perhaps we should require that **FilePosInt** be abstract; that is, `abstraction FilePosInt : INTEGER = Int`, so that programmers are forced to write more portable code.

Redirecting IO streams

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).

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

$s =$ **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.

$f =$ **mk_outstream**(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 an **IO.outstream** and a **StreamIO.outstream** is that the former may be redirected using **set_outstream**. Think of the former as a `ref` of the latter.

$s =$ **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

```
  val f' = mk_outstream(get_outstream f)
```

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

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 **std_out**, 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.

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; but for more sophisticated users, the translation functions are made visible as **TextIO.translate_in and TextIO.translate_out. On Unix systems, these will be identity functions. See section 9.4.**

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.[6]

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

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

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

[6]Perhaps **Io** shouldn't be in this signature; where should it live?

```
              close      : unit -> unit,
              getpos     : (unit -> pos) option,
              setpos     : (pos -> unit) option,
              endpos     : (unit -> pos) option,
              find_pos   : ({data: vector, first: int, nelems: int}*pos
                               -> pos)) option}

    datatype writer = Wr of
             {write_noblock: ({data: vector, first: int, nelems: int} ->
                                int option) option,
              writea_noblock: ({data: array, first: int, nelems: int} ->
                                 int option) option,
              write_block: ({data: vector, first: int, nelems: int} ->
                              int) option,
              writea_block: ({data: array, first: int, nelems: int} ->
                               int) option,
              block: (unit->unit) option,
              can_output: (unit->bool) option,
              name: string,
              chunksize: int,
              close: unit -> unit,
              getpos : (unit->pos) option,
              setpos : (pos->unit) option,
              endpos : (unit->pos) option}

    val open_in: string -> reader
    val open_out: string -> writer

    val augment_in : reader -> reader
    val augment_out: 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 to this reader are illegal and must be checked for by the reader (the Io exception must be raised).

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,first=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,first=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 reamain 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. Useful even for non-seekable files, if the endpos 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.

endpos()

(optional) The position at the end of the file.

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

(option) If find_pos=NONE, then positions in the reader must correspond 1–1 to elements returned from the read functions; and endpos returns exactly the number of elements in the file. If find_pos=SOME(f), then this is not necessarily true. In that case, $f(\text{data} = v, \text{first} = i, \text{nelems} = n, p)$ tells the position 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.

One of read_block, reada_block, read_noblock, or reada_noblock must be provided.

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

1. Absence of all of read_block, reada_block, and block means that blocking input is not possible.

2. Absence of all of read_noblock, reada_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.

The augment_in function takes a reader r and produces a reader in which as many as possible of read_block, reada_block, read_noblock, reada_noblock are provided, by synthesizing these from the operations of r.

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. 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.
9. If getpos is provided, and *pos* positions do not correspond 1–1 to elements returned from the read functions, then find_pos must be provided. This will typically be necessary when the reader is performing some sort of translation on the input stream. If the translation function is invertible, then find_pos will be straightforward to implement. If not invertible, then find_pos can seek to *pos* in the underlying file, and re-translate forward to the right point. In that case, the implementation of find_pos will probably require: $p_0 =$ getpos, setpos(*pos*), read, setpos(p_0) to restore the file position to what it was before the find_pos operation.
10. If find_pos is SOME(f), then clients of this reader will probably do many getpos operations. For efficiency, if find_pos=SOME(f), the reader should keep track of file positions, incrementing with each read, so as to avoid a system call for each getpos operation.
11. Even when find_pos=NONE, the reader could keep track of file positions to avoid system calls for getpos. It may be the case, however, that the vast majority of files are read only sequentially, so it may not be useful to optimize getpos.

The components of a writer are:

write_noblock{buf=v,first=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 SOME(k); or (if the write would block) returns NONE. $k = 0$ is not recommended (prohibited?). Raises Io on failure of underlying system call, or Subscript if $i < 0$ or $i + n > \mathit{length}(v)$.

writea_noblock{buf=a,first=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 SOME(k); or (if the write would block) returns NONE. $k = 0$ is not recommended (prohibited?).

write_block{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.

writea_block{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.

write_noblock{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.

writea_noblock{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.

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

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

chunksize
: The recommended (efficient) size of write operations on this writer. This is typically to the block size of the operating system's buffers. If that is not known, a value of 2048 or 4096 will

probably work well. Chunksize = 1 strongly recommends (but cannot guarantee, since buffering occurs in other modules, not this one) unbuffered I/O on the writer. Chunksize ≤ 0 is illegal (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 and must be checked for by the writer (Io must be raised).

getpos()
: (optional) Tells the current position within the file. Most useful on seekable writers.

endpos()
: (optional) The position at the end of the file.

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

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

1. Absence of all of write_block, writea_block, and block means that blocking output is not possible.
2. Absence of all of write_noblock, writea_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 writea_noblock, write_noblock, and can_output from a writer means that nonblocking output is impossible. But the standard StreamIO modules do not support nonblocking output anyway.
6. Absence of getpos means that buffered setpos may be less efficient.
7. If $pos \neq int$ then buffered setpos may be difficult to implement; but for "standard" modules $pos{=}int$

8. Absence of setpos prevents random access.

The augment_out function takes a writer w and produces a writer in which as many as possible of write_block, writea_block, write_noblock, writea_noblock are provided, by synthesizing these from the operations of w.

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

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

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

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

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

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

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

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
               val someElem : A.elem
               structure Pos : INTEGER) : PRIM_IO =
  struct . . . end
```

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

```
fun augment_in (r as Rd r') =
   let reada_to_read reada i =
            let val a = A.array(i,someElem)
                val i' = f{data=a,first=0,nelems=i};
             in A.extract(a,0,i')
            end

       val read_block' =
          case r
           of Rd{read_block=SOME f,...} => SOME f
            | Rd{reada_block=SOME f,...} => SOME(reada_to_read f)
            | Rd{read_noblock=SOME f,block=SOME b,...} =>
                             SOME(fn i => (b(); f i))
            | Rd{reada_noblock=SOME f, block=SOME b,...} =>
                             SOME(fn i => (b(); reada_to_read f i))
            | _ => NONE
     . . .
    in Rd{block= #block r', . . . read_block=read_block', . . . }
   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 pos    sharing type pos = PrimIO.pos

    type instream
    type outstream

    val open_in : string -> instream
    val mk_instream   : PrimIO.reader -> instream
    val close_in      : instream -> unit
    val setpos_in     : instream * pos -> instream
    val getpos_in     : instream -> pos
    val endpos_in     : instream -> pos
    val input         : instream -> vector * instream
    val input_all     : instream -> vector
    val input_noblock : instream -> (vector * instream) option
    val input1        : instream -> elem option * instream
    val input_n       : instream * int -> vector * instream
    val end_of_stream : instream -> bool
    val get_reader    : instream -> PrimIO.reader

    val open_out: string -> outstream
    val mk_outstream : PrimIO.writer -> 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 -> pos
    val setpos_out  : outstream * pos -> unit
    val endpos_out  : outstream -> pos
    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.

$f =$ **open_in**(s)

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

$f =$ **mk_instream**(r)

Create a buffered stream f **from a reader** r. **In** r, **read_block, reada_block, and block must not all be** NONE **or an Io exception will be raised. (Most users will normally use open_in instead.)**

close_in(f)

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

$g =$ **setpos_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 of f. *Not always supported.*

endpos_in(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.**

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

Return the vector v **of all the elements of** f **up to end of stream. Semantically equivalent to:**

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

where ^ is the concatenation operator on element vectors.

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

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

$(c, f') =$ **input1**(f)

If at least one element e of f is available, return (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
```

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

```
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 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)`.

get_reader(f)

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

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

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

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

$f = $ **mk_outstream**(w, s)
Create a buffered outstream f from a writer w. In w, write_block, writea_block, and block must not all be NONE or an Io exception will be raised.

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

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

output(f, v)
Write the sequence v to f; this may block until the system is prepared to accept more output. StreamIO does not provide any nonblocking output function.

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

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.*

endpos_out(f)
The position at the end of file f. *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, ...}.[7]

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 input_noblock 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 = open_in name (* or,  a = mk_instream(r)  *)
 in close a;
    size(input a) = 0
end
```

[7] Should we make a special OS.notSupported?

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; input_noblock f) of SOME _ => true | NONE => false)
```

The end_of_stream test is equivalent to input returning an empty sequence:

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

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

```
 let val f = mk_instream(reader)
     val (a,f') = input(f,n)
     val PrimIO.Rd{chunksize,...}=get_instream f
  in chunksize>1 orelse end_of_stream 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 = 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 orelse end_of_stream 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 end_of_file will not make repeated system calls.

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

```
val monitor: reader -> {rd: reader,
                        chars_read: int ref,
                        op_count: int ref}
```

and making statements such as:

```
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 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 = 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.

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 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
               sharing type PrimIO.pos=FilePosInt.int
               ) : 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., writea_block 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 try again. If the first or any successive write attempt returns zero elements written (or raises an exception) then flush_out raises an Io exception.

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.

With buffered reading, a getpos operation on the instream may be done in the middle of a buffer. If find_pos is NONE in the underlying reader, then the StreamIO.getpos can be implemented by asking the current position (of the end of the buffer) and then subtracting. But find_pos is SOME(f), then the subtraction won't work, because elements don't correspond 1–1 to positions. In that case, it will be necessary to call find_pos; this in turn requires the position at the beginning of the buffer. But this means that the StreamIO system must do a getpos just before reading each new buffer, and remember the buffer position.[8] (This is *not* necessary if find_pos=NONE.)

Here are some suggestions for efficient performance:

[8]This is rather unfortunate.

- 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_noblock operation of the underlying reader can replenish the buffer when necessary.
- Each reader may provide only a subset of read_block, read_noblock, block, can_input, etc. An augmented reader that provides more operations that can be constructed using PrimIO.augment_in; 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 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 endpos, subtract the current position, and try to read that much. But one should also keep things rounded to the nearest chunksize.
- The use of endpos to try to do (large) read operations of just the right size will be inaccurate if find_pos=SOME. 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 writea_block. Very large outputs can be written directly from the argument string using write_block.
- But how should the current buffer position be remembered? Either a getpos every time endpos 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 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 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.

8 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
   type pos = S.pos
   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 !
   val endpos_in = S.endpos_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 !
   val endpos_out = S.endpos_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 STREAM_IO and IO. The only purpose of the extra ref in 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.[9]

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))
    in TextIO.setpos_out(out,some_pos);
       output(out,"Hello ");
       output(out,"World\n");
       flush_out out
   end
   ```

[9]The example here uses two file descriptors on the same file. This works fine on Unix; I'm not sure about other operating systems. Perhaps we should have a **PrimIO.open_in_out** function that produces a reader and a writer sharing a file descriptor (as a supplement to **PrimIO.open_in** and **PrimIO.open_out**). Or perhaps this function should be in some nonstandard module; it doesn't have to live inside the PrimIO module.

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

   ```
   let val inf = TextIO.mk_instream(TextIO.StreamIO.mk_instream(reader))
    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.**

9.2 Other reader/writer devices

The functions open_in and open_out provide system-default ways to create readers from "file names."

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

```
structure Socket :
   sig  type socket_name
        structure P = TextIO.StreamIO.PrimIO
        val open_socket_text_reader: socket_name -> P.reader
        val open_bidirectional_socket: socket_name ->
                    P.reader * P.writer
        . . .
   end
```

Then the user could buffer these readers by using mk_instream.

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

```
structure Socket :
   sig  type socket_name
        val open_socket_text_in: socket_name -> TextIO.instream
        val open_bidirectional_socket: socket_name ->
                    TextIO.instream * TextIO.outstream
        . . .
   end
```

and the user could extract the reader by using get_instream and get_reader.

Some operating systems have a notion of open for append; this differs from open_out followed by setpos(endpos(f)) in that if *other* processes are also appending to the file, each successive write has an implicit setpos to end of file. An operating-system support module could provide a way to create an appending writer.[10]

[10] Or should this be a standard feature of **TextIO**?

9.3 String readers/writers

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

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

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

```
 fun string_reader(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{read_noblock = SOME(fn n => SOME(read n)),
     reada_noblock = NONE,
     read_block = SOME(read),
     reada_block= NONE,
     block = SOME(fn()=>()),
     can_input = SOME(fn()=>true),
     name="<string>",
     chunksize=size source,
     close=fn()=>(),
     getpos=SOME(fn()=>!pos),
     setpos=SOME(fn k => if 0<=k andalso k <= size source then pos:=k
                   else raise Io{ml_op="setpos",name="<string>",os_op="",
                                 reason="position out of bounds",
                                 syserror=OS.noError}),
     endpos=SOME(fn()=>size source)}
  end

 val open_string : string -> instream =
       TextIO.mk_instream o TextIO.StreamIO.mk_instream o string_reader
```

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{
     read_noblock = option (fn get =>  option stringCR o get)
                         (#read_noblock rd,)
     reada_noblock = (* etc. *),
     read_block = option (fn get => stringCR o get) (#read_block rd),
     reada_block=  (* etc. *),
     block = #block rd,
     can_input = #can_input rd,
     name= #name rd,
     chunksize= #chunksize rd,
     close= #close rd,
     getpos=#getpos rd,
     setpos=#setpos rd,
     endpos=#endpos rd,
     find_pos= (case (#getpos rd, #setpos rd,
                      TextIO.StreamIO.PrimIO.augment_in rd0)
                 of (SOME get, SOME set,
                     TextIO.StreamIO.PrimIO.Rd{read_block=readb,...}) =>

                     SOME(fn ({data,first,nelems},pos)=>
                            let val p0 = get()
                               val p1 = (set(pos);
                                         retranslate(readb,nelems,pos))
                            in set(p0); p1
                           end)
                     | _ => 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, either find_pos must be provided, or getpos, setpos, and endpos must not be provided. Here we have chosen to provide find_pos whenever possible. Because the translation is not invertible (we don't know where the CR characters might have been), find_pos 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 mk_instream).

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*, perhaps 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 MyPrimIO : PRIM_IO =
  sig  type pos = string (* or whatever *)
       datatype reader = Rd  of ....
```

```
      ...
  end
```

Now one can write translated readers that can deal with translated positions more flexibly, since there's no 1–1 correspondence property that must be maintained.

The only problem is that the standard StreamIO functor cannot be applied, because the sharing constraint `type pos=int` **is violated.**[11]

The user can write his own buffering functor:

```
functor MyStreamIO(structure PrimIO : PRIM_IO ...
                   (* not sharing type PrimIO.pos=int *)
                  ) : STREAM_IO = struct ... end

structure MyIO = MyStreamIO(structure PrimIO = MyPrimIO ...)
```

Now MyIO.instream is a different type than TextIO.StreamIO.instream. If one didn't rely on `pos=int`**, then one could still make use of the MyIO interface:**

```
functor MyApplication(IO : STREAM_IO) = struct ... end
```

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

```
 fun standardize (MyPrimIO.Rd rd) =
     TextIO.StreamIO.PrimIO.Rd{
         read_noblock = #read_noblock rd,
         reada_noblock = #reada_noblock rd,
         read_block = #read_block rd,
         reada_block=  #reada_block rd,
         block = #block rd,
         can_input = #can_input rd,
         name= #name rd,
         chunksize= #chunksize rd,
         close= #close rd,
         getpos=NONE,
         setpos=NONE,
         endpos=NONE}
```

[11]The sharing of *pos=FilePosInt.int* is useful to clients of **StreamIO**. Perhaps it is not necessary for the internals of the functor. If that were the case, then it would not be necessary to define the functor **MyStreamIO**, because functor **StreamIO** could be used. But **MyIO** would still be incompatible with **TextIO.StreamIO**.

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 input_line function, that reads up to the next newline character. A naive implementation would read characters, then concatenate them:

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

Now, we may wish to avoid all the list construction and implode call.[12] Thus:

```
fun input_line (f: TextIO.instream) =
 let val g0 = TextIO.get_instream f
     fun loop(i,g) = case input1 g
                   of (SOME(#"\n"),_) => i+1
                    | (SOME c, g') => loop(i+1,g')
                    | (NONE,_) => i
  in TextIO.input_n(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 input_n call.[13] 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.

[12]Is this still called **implode** in the new basis?

[13]There is some building of **SOME** constructors. We must still discuss whether input1 should return a char option, or just return a char with exception on end of file. But that's a separate issue. However, in the **input_line** function I show here, raising the exception leads to natural, efficient behavior if we also change the semantics of **input_line** to say that an exception should be raised if the last line of the file does not end with a newline character.

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 input_n to extract it and advance the stream.

10 Loose ends

What about opening files for append?

What about user (and other) interrupts during buffered I/O operations?