A Guide to SML2C

David R. Tarditi Jr. and Anurag Acharya
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
tarditi@cs.cmu.edu

August 1, 1992

(c) 1991, 1992 Carnegie Mellon University

1. Introduction

sml2c is a Standard ML compiler which compiles an extended version of Standard ML to portable C code. The extensions include first-class continuations[2], asynchronous signal handling [5], support for shared-memory multiprocessing [4], separate compilation and a facility to freeze and restart programs. It is based on version 0.75 of the Standard ML of New Jersey (SML/NJ) implementation[1] and shares its front-end and most of the runtime system. It has been developed at the School of Computer Science, Carnegie Mellon University as part of research into portable implementation of properly tail-recursive languages with first-class continuations [6].

sml2c is a batch compiler and compiles only module-level declarations, i.e. signatures, structures and functors. It provides the same pervasive environment for the compilation of these declarations as SML/NJ. As a result, module-level programs that run on SML/NJ can be compiled by *sml2c* without any changes. It does not support SML/NJ style debugging and profiling. Support for profiling may, however, be added in future.

sml2c is highly portable and has been ported to the IBM RT, the Decstation 3100, the Decstation 5000, the Vax, the Omron Luna 88k, the Sun-3, the Sun-4, the Sequent Symmetry, and the Intel-486 architectures. The generated code is highly portable and makes very few assumptions about the structure of the machine. The runtime system, which it shares with SML/NJ, however, has several operating-system dependencies. *sml2c* has eliminated some of these dependencies by using portable schemes for garbage collection and for freezing and restarting a program[6].

The remainder of this report is organized as follows. Section 2 describes the installation procedure for *sml2c*. It also describes the portability requirements for *sml2c*. Section 3 is a users' manual to the compiler. It describes the available options and other pragmatic issues. Section 4 contains a brief list of the of the differences between this implementation of Standard ML and the language described in the *Definition of Standard ML* [3]. See Chapter 5 of the release notes for SML/NJ (found in doc/release-notes) for more information. Section 5 describes the standard library that is available as a pervasive environment. Finally, section 6 describes the separate compilation facility.

- The machine must be word-addressable with 32-bit words.
- All pointers must fit into a single word and must be word-aligned.
- Integers must be 32-bit quantities in a two's-complement representation.
- The right shift operator `>>` must use sign-extension to fill the high-order bits of a shifted word.
- The native character representation must be ASCII.

Figure 1: Primary requirements for porting *sml2c*

- Values of type `double` must be 8 bytes in size.
- The C math library must meet the ANSI C specifications for math functions
- Floating point values must be word-aligned.

Figure 2: Requirements for floating point operations.

2. Installation

This section describes the installation procedure and the portability requirements for *sml2c*. The first subsection lists the assumptions that the *sml2c* compiler makes about the target machine and the operating system dependencies that the runtime system contains. The second subsection describes the installation procedure.

2.1. Portability Requirements

There are two sets of assumptions that the *sml2c* compiler makes about the target machines and the C compilers on those machines. The first set of assumptions defines the primary requirements that must hold before *sml2c* can be bootstrapped onto a machine. These are listed in Figure 1. The requirements are fairly lax and are satisfied by a large number of machines including most recently introduced machines. The second set of assumptions is for ensuring correctness of floating point operations. The corresponding requirements have been listed in Figure 2. These are also satisfied by a large number of machines.

sml2c uses most of the SML/NJ runtime system. The SML/NJ runtime system has operating system dependencies which limit the set of operating systems under which *sml2c* can be run. The operating systems supported by the SML/NJ runtime system are listed in Figure 3.

- Unix 4.3 BSD
- Mach emulating Unix 4.3 BSD
- Ultrix
- SunOS
- RISCos
- More
- V9 or V10

Figure 3: Operating systems on which *sml2c* may be installed

2.2. Installation Procedure

There are two stages in the *sml2c* installation procedure. The first stage builds the runtime system and the second builds the compiler. The runtime system is built by changing to the `src` directory in the distribution and running the following command:

`makeml -c` $< os >< machine >< ccom >$

The value of *os* can be one of `mach,ultrix,sunos,v9,v10,riscos, more, bsd`. The value of *machine* can be one of `next, NeXT, sun3, sun4,vax,decstation,DECstation,sgi,mips`. The value of *machine* can also be `unknown` for machines not listed. The operating systems on which we have successfully tested *unknown* are Mach and BSD Unix. The *unknown* option does not work with HP/UX. We have not tested *unknown* on other operating systems. Specifying *ccom* is optional. It specifies an alternative C compiler to be used to compile the C files. The only value currently supported is `-gcc` (which instructs `makeml` to use the gcc compiler). The gcc compiler is recommended for Sun3 and Vax machines.

The `makeml` command will generate the runtime system library and install it as `runtime/libsml2c.a`. Building the compiler will require access to this library. Therefore, it is necessary to make sure that the C compiler is able to find it. In a typical Unix environment, this can be done by appending the path to the runtime system library to the `LPATH` environment variable.

The *sml2c* compiler is built by using the makefile that is available in the `src` directory. The `make` takes quite a while (from an hour on a Sparc 1 to several hours on a Sun 3) and generates an executable file called `sml2-comp` which is the bare-bones version of the compiler, i.e. it does not contain the symbol table information for the pervasive environment. This information is generated by running `sml2c-comp` without any arguments. This results in the generation of the file `sml2c-heap` shich contains the symbol table information. This completes the building of the *sml2c* compiler.

For convenience, it is expected that the local maintainers will install the compiler (`sml2c-comp`), its heap (`sml2c-heap`), and the library (`runtime/libsml2c.a`) to wherever they are supposed to

live locally. If the library is copied, it may be necessary to run `ranlib` on it. The compiler is supposed to be run with the command:

```
sml2c-comp -s sml2c-heap ...
```

It is expected that the users will be provided with a seamless interface to the compiler which hides the location of the heap file. In typical Unix environments, this can be done by creating a shell script called `sml2c`. For example, if the compiler and its heap are in /usr/local/bin, an appropriate shell script would be:

```
#!/bin/sh
/usr/local/bin/sml2c-comp -s /usr/local/bin/sml2c-heap $@
```

3. Using sml2c

sml2c is a batch compiler. Unlike SML/NJ, it does not support interactive compilation. The format of the command line arguments is like most other Unix compilers. For example, the following command

```
sml2c -o a a.sml b.sml
```

compiles the files `a.sml` and `b.sml` and generates an executable file called `a`. *sml2c* compiles the `.sml` files specifed in the sequence they are specified on the command line. Each file is compiled to a C file. After all files have been compiled to C, a C compiler is invoked to generate the object files and link them with the runtime system library to produce the executable file. *sml2c* removes all the intermediate files (C and object code) that it generates during its operation. Several options are available to guide and control the operation of the compiler. These are described in Section 3.2.

3.1. Setting up your environment

You must set up your environment so that the C compiler can find the runtime system library automatically. On most Unix systems, this can be done by adding the directory containing the runtime system library to the list of directories associated with the `LPATH` environment variable.

3.2. Options

sml2c supports several options and optimization flags. Any string not recognized as an option or an optimization flag is assumed to be the name of a source file. *sml2c* understands the following options:

`-b` *c*: Use *c* as the C compiler name

`-c` *m*: Generate only C files and do not invoke the C compiler. Create a makefile *m*, for Unix make, that can be used to compile the C files. This is useful for cross-compiling C code or using C compilers with different interfaces.

`-cf` *f*: Pass *f* as a flag to the C compiler.

`-gcc`: Use gcc instead of cc. Recommended for Sun-3's and Vaxes

`-o` *f*: Create an executable file named f. This option must always be specified.

`-pervasives`: Generate new C files for the pervasive environment.

`-plain`: Do not pass the `-O` flag to the C compiler. Useful when bugs in an optimizer cause a C compiler to fail.

`-statistics`: Print (static) statistics for various optimizations after compilation finishes.

`-timing`: Time how long the C compiles take.

3.3. Optimization flags

There are a number of flags which control the optimizations that *sml2c* performs. The general form is `-f`*flag*. Flags have postive and negative forms; the negative form of `foo` is `no-foo`. The negative form disables the flag and is used to override flags which are enabled by default. Here are the flags:

`O`: Allow aggressive optimization of programs by the SML/NJ optimizer. This flag is disabled by default.

`signals`: Provide full support for signals by disabling optimizations which are incompatible with the asynchronous signal handling mechanism. This flag is enabled by default — that is, signals are supported by default. You can enable all these optimizations if you are not using signals to get significantly faster (20-40% faster) code using the flag `no-signals`.

`arithmetic`: Optimize overflow checks for arithmetic operations. This flag is enabled by default.

`function-integration`: If all call paths to a known function f are dominated by another function g, we can integrate f into the body of g. This flag speeds up inner loops. It is disabled by default, since it is incompatible with the asynchronous signal handling mechanism.

`local-variables`: Use local variables to shadow psuedo-register values within the C functions that it generates. If this flag is disabled, nothing will end up in machine registers. The flag is enabled by default.

`tail-call-elimination`: It eliminates passes by directly tail-recursive functions through the dispatch loop. This flag is enabled by default.

`unsafe-arithmetic`: This flag disables overflow checking on arithmetic operations. The use of this flag is strongly discouraged. This flag is disabled by default.

`zero-check`: Keep heap checks where no allocation occurs. The negative form is incompatible with signals, but results in faster code. The flag is enabled by default.

3.4. Managing memory use

SML provides automatic storage management through a garbage-collected heap. Since the heap is used intensively, the choice of heap size can have a significant impact on performance. A program determines an efficient heap size automatically on startup, resizes the heap up or down as the amount of live data changes, and complains if it runs out of memory.

There are three runtime system parameters that help determine the size of the heap. The following command line arguments control these parameters for *sml2c* and all programs produced by it:

`-r` j: the desired ratio between the size of live data and the size of the heap is set to j, a positive integer greater than or equal to 3. The default is $j = 5$

`-m` k: set the soft maximum heap size (softmax) to k Kbytes (k a positive integer), with the default being $k = 4096$ (4M) You should increase this if you have a machine with more than 8MB.

`-g` l: set the garbage collection message level to l, where l is 0,1,2, or 3

The `-g` option specifies how much information is printed about garbage collections. The value $l = 0$ suppresses all garbage collection messages, l=1 causes messages to be printed for major collections, $l = 2$ adds messages for heap resizing, and $l = 3$ adds messages for minor collections and failure to maintain the desired ratio. The default is $l = 0$.

In general, the larger the heap size, the lower the garbage collection overhead. In principle, heap size is limited by the total virtual memory available to a process. In practice, since heavy paging will slow down a program drastically, it is very desirable to keep heap size within the bounds of available physical memory, taking into account the memory requirements of other processes sharing the physical memory. Setting a high ratio will cause the compiler to be aggressive in its use of memory, while setting a reasonable limit on heap size can prevent excessive paging. The maximum heap size can be controlled externally by using the "limit datasize" command in the C shell. For instance, on an 8MB Sun 3 you might execute the shell command

```
limit datasize 6M
```

before running a program. This would prevent the heap from growing larger than 6 MB regardless of the value of the ratio. This is a hard limit, so it might cause a program to run out of memory and fail even though plenty of virtual memory is available.

The soft maximum heap size (softmax) specified with the `-m` option is a less drastic way of controlling the system's appetite for memory. The heap will grow larger than softmax only when necessary to maintain a ratio of 3. Thus when softmax is reached, the heap will not grow until the size of live data is one third of softmax, and thereafter it will grow more slowly than with the default ratio of 5 (or the user defined ratio). A reasonable strategy is to set softmax to the maximum amount of physical memory that you know will be available and set ratio to a high value (e.g. 20), so that the heap size will tend to stick at softmax, thereby taking advantage of available memory.

The ratio and softmax parameters can also be changed at any time from within a program since they are the values of the variables

```
System.Control.Runtime.ratio: int ref,
System.Control.Runtime.softmax: int ref.
```

As an example,

```
sml2c ...  -r 5 -m 8000
```

might be appropriate when running *sml2c* on a machine with with 16MB of physical memory.

3.5. Freezing and Restarting Programs

The functions `exportFn` and `exportML` produce heap files containing the state of the program and the environment. Heap files are stamped with a unique identifer that prevents them from being used with executables other than the one that generated it.

A frozen program `foo` with a heap file `foo.heap` can be restarted by the following command:

```
foo -s foo.help
```

All executables generated by *sml2c* support this interface.

4. Summary of Differences

Here is a list of many of the differences between Standard ML of New Jersey and the Definition of Standard ML[3]. See Chapter 5 of the release notes for SML/NJ (found in doc/release-notes) for more information.

- First-class continuations are supported.
- Multiple specifications of a name within a name space in a signature are prohibited.
- `local` and `open` specs in signatures have a different semantics (see the file doc/localspec)
- Different right-associative operators of the same precedence associate to the right in SML/NJ.
- The symbol `=` can be re-bound (though usually you get a warning message)
- The construct `val ... and rec ...` is not permitted; the `rec` must immediately follow the `val`.
- The Definition prohibits some, but not all, signatures that could never be matched by any structure. SML/NJ is more liberal about such signatures.
- The initial basis is a superset of the one described in the Definition. It is described in detail in Section 5.
- The `@` operator is right-associative in SML/NJ.
- The arithmetic overflow exceptions are all identified together. `Sum`, `Prod`, `Diff`, `Neg`, `Exp`, `Floor` are all equivalent to `Overflow`. `Div` and `Mod` are equivalent, and distinct from `Overflow`. `Ln` is a distinct exception; `Sqrt` is a distinct exception.
- Strings carried by the `Io` exception are more informative in SML/NJ.
- More efficent implementations of integer division and modulus are provided by the `quot` and `rem` functions. These functions implement round-towards-zero versions of integer division modulus. This differs from the rounding behavior for `div` and `mod` specified in the Definition.
- Curried versions of input and output are available as the functions `inputc` and `outputc`. The function `inputc` does not block if fewer than the requested n characters are available, unlike `input`.

5. The standard library

A standard set of values, types, exceptions, etc. are *pervasive*—they are in the initial environment and available in every structure[1] This *standard library* is grouped into structures; each structure deals with the operations on one or two abstract types. The signature of each of these modules, with an informal explanation of the semantics, is given in this section.

```
signature GENERAL =
  sig
    infix 3 o
    infix before
    exception Bind
    exception Match
    exception Interrupt
   (* NOTE: Interrupt is never raised by the system, but is
      included to provide some compatibility with the
      Definition. *)

    type 'a cont
    val callcc : ('1a cont -> '1a) -> '1a
    val throw : 'a cont -> 'a -> 'b

    val o : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)
    val before : ('a * 'b) -> 'a

    type exn
    type unit
    infix 4 = <>
    val = : ''a * ''a -> bool
    val <> : ''a * ''a -> bool
  end
abstraction General : GENERAL
```

The structure `General` contains various miscellaneous and general-purpose values, types, and exceptions. The infix `o` is the function composition operator. The infix `before` evaluates both of its arguments and returns the first one. The exceptions `Bind` and `Match` are automatically raised by patterns that fail to match. The exception `Interrupt` is never raised by the system, but is provided for compatibility with the Definition. In the Definition, the exception `Interrupt` is supposed to be raised when an Interrupt signal is received by the program (e.g. when the user types Control-C or its equivalent). This has been superceded by an extension to Standard ML for handling asynchronous signals. This extension is described in [5]. The functions `callcc` and `throw` and the type `'a cont` are used for explicit manipulation of continuations. Their use is explained in [2]. The

[1] This section has been taken from from the *Standard ML Reference Manual (Preliminary)*, which is distributed with Standard ML of New Jersey system. It has been updated to reflect the pervasive environment for this version of the **sml2c** compiler. The author, of course, are solely responsible for any errors in this section.

standard type `exn` is the type of all exceptions and `unit` is the type of empty records. The `=` and `<>` operators are defined here *pro forma*.

5.1. List

```
signature LIST =
  sig
    infixr 5 :: @
    datatype 'a list = :: of ('a * 'a list) | nil
    exception Hd
    exception Tl
    exception Nth
    exception NthTail
    val hd : 'a list -> 'a
    val tl : 'a list -> 'a list
    val null : 'a list -> bool
    val length : 'a list -> int
    val @ : 'a list * 'a list -> 'a list
    val rev : 'a list -> 'a list
    val map :  ('a -> 'b) -> 'a list -> 'b list
    val fold : (('a * 'b) -> 'b) -> 'a list -> 'b -> 'b
    val revfold : (('a * 'b) -> 'b) -> 'a list -> 'b -> 'b
    val app : ('a -> 'b) -> 'a list -> unit
    val revapp : ('a -> 'b) -> 'a list -> unit
    val nth : 'a list * int -> 'a
    val nthtail : 'a list * int -> 'a list
    val exists : ('a -> bool) -> 'a list -> bool
  end
```

The semantics of this module are defined by the following implementation.

```
abstraction List: LIST =
  struct
    infixr 5 :: @
    infix 6 + -
    datatype 'a list = :: of ('a * 'a list) | nil
    exception Hd
    fun hd (a::r) = a | hd nil = raise Hd
    exception Tl
    fun tl (a::r) = r | tl nil = raise Tl
    fun null nil = true | null _ = false
    fun length nil = 0 | length (a::r) = 1 + length r
    fun op @ (nil,l) = l | op @ (a::r, l) = a :: (r@l)
    fun rev l = let fun f (nil, h) = h
                      | f (a::r, h) = f(r, a::h)
```

```
              in  f(l,nil)
              end
    fun map f = let fun m nil = nil | m (a::r) = f a :: m r
              in  m
              end
    fun fold f = let fun f2 nil = (fn b => b)
                       | f2 (e::r) = (fn b => f(e,(f2 r b)))
                 in  f2
                 end
    fun revfold f l = fold f (rev l)
    fun app f l = (map f l; ())
    fun revapp f l = app f (rev l)
    exception Nth
    fun nth(e::r,0) = e
      | nth(e::r,n) = nth(r,n-1)
      | nth _ = raise Nth
    exception NthTail
    fun nthtail(e,0) = e
      | nth(e::r,n) = nthtail(r,n-1)
      | nth _ = raise NthTail
    fun exists f =
          let fun g nil = false | g (h::t) = f h orelse g t
          in  g
          end
  end
```

5.2. Array

```
signature ARRAY =
  sig
    type 'a array
    exception Size
    exception Subscript
    val array : int * 'a -> 'a array
    val arrayoflist : 'a list -> 'a array
    val tabulate: int * (int -> 'a) -> 'a array
    val sub : 'a array * int -> 'a
    val update : 'a array * int * 'a -> unit
    val length : 'a array -> int
  end
```

Arrays may be made whose elements are any type. `array(n,x)` returns a new array of n elements, indexed from 0 to $n-1$, initialized to x. `a sub i` returns the i^{th} element of the array a. `update(a,i,z)` sets the i^{th} element of the array a to the value z. `tabulate(n,f)` creates a new array of n elements, indexed from 0 to $n-1$, whose i^{th} elements are each initialized to the value of $f(i)$.

Two arrays are equal if and only if they are the same array (created with the same call to `array`); except that all arrays of length 0 may be equal to each other, depending on the implementation.

Note that, by default, the Array structure is present but not opened in the initial environment. The declaration `open Array` may be used to open it.

The following implementation defines the semantics of arrays, though in practice arrays are implemented much more efficiently.

```
abstraction Array : ARRAY =
  struct
   type 'a array = 'a ref list
   exception Size
   exception Subscript
   fun array(0,x) = nil | array(n,x) = ref x :: array(n-1,x)
   fun arrayoflist l = map ref l
   fun tabulate (i,f) =
         if i<0 then raise Size
         else let fun loop j = if j<i then ref(f j) :: loop (j+1) else nil
              in loop 0
              end
   fun sub(a,i) = !(nth(a,i)) handle Nth => raise Subscript
   fun update(a,i,z) = nth(a,i) := z handle Nth =>raise Subscript
   fun length a = List.length a
  end
```

5.3. Input/Output

```
signature IO =
  sig
    type instream
    type outstream
    exception Io of string
    val std_in : instream
    val std_out : outstream
    val std_err : outstream
    val open_in : string -> instream
    val open_out : string -> outstream
    val open_append : string -> outstream
    val open_string : string -> instream
    val close_in : instream -> unit
    val close_out : outstream -> unit
    val output : outstream * string -> unit
    val outputc : outstream -> string -> unit
    val input : instream * int -> string
    val inputc : instream -> int -> string
```

```
    val input_line : instream -> string
    val lookahead : instream -> string
    val end_of_stream : instream -> bool
    val can_input : instream -> int
    val flush_out : outstream -> unit
    val is_term_in : instream -> bool
    val is_term_out : outstream -> bool
    val set_term_in : instream * bool -> unit
    val set_term_out : outstream * bool -> unit
    val execute : (string * string list) -> instream * outstream
    val execute_in_env : (string * string list * string list) -> instream * outstream
    val exportML : string -> bool
    val exportFn : string * (string list * string list -> unit) -> unit
    val use : string -> unit
    val use_stream : instream -> unit
  end
structure IO : IO
```

Here are the basic functions and types one needs to know to understand the operation of the I/O structure. The type `instream` is the type of input streams and the type `outstream` is the type of output streams. The exception `Io` is used to represent all of the errors that may arise in the course of performing I/O. The value associated with this exception is a string representing the type of failure. In general, any I/O operation may fail if, for any reason, the host system is unable to perform the requested task. The value associated with the exception should describe the type of failure, insofar as this is possible.

The standard prelude binds `std_in` to an instream and `std_out` to an outstream. For interactive ML processes, these are expected to be associated with the user's terminal. However, an implementation that supports the connection of processes to streams may associate one process's `std_in` to another's `std_out`.

The `open_in` and `open_out` primitives are used to associate a disk file with a stream. The expression `open_in(s)` creates a new instream whose producer is the file named `s` and returns that stream as a value. Similarly, `open_out(s)` creates a new `outstream` associated with the file `s`, and returns that stream.

The `input` primitive is used to read characters from a stream. Evaluation of `input(s,n)` causes the removal of `n` characters from the input stream `s`. If fewer than `n` characters are currently available, then the ML system will block until they become available from the producer associated with `s`[2]. If the end of stream is reached while processing an `input`, fewer than `n` characters may be returned. Attempting `input` from a closed stream raises `Io`.

There are two new functions, `outputc` and `inputc`. `outputc` is a curried version of `output`, while `inputc` is similar to `input`. The function

[2]The exact definition of "available" is implementation-dependent. For instance, operating systems typically buffer terminal input on a line-by-line basis so that no characters are available until an entire line has been typed.

```
inputc f n
```

returns the empty string on an end-of-stream. Otherwise it returns a string of length k, $0 < k \leq n$, depending on what is available and convenient. This differs from the behavior of input, which will block until n characters become available.

The function `lookahead(s)` returns the next character on `instream s` without removing it from the stream. Input streams are terminated by the `close_in` operation. This primitive is provided primarily for symmetry and to support the re-use of unused streams on resource-limited systems. The end of an input stream is detected by `end_of_stream`, a derived form that is defined as follows:

```
fun end_of_stream(s) = (lookahead(s)="")
```

Characters are written to an `outstream` with the `output` primitive. The string argument consists of the characters to be written to the given outstream. The function `close_out` is used to terminate an output stream. Any further attempts to output to a closed stream cause `Io` to be raised.

In addition to the basic I/O primitives, provision is made for some extensions that are likely to be provided by many implementations. The function `execute` is used to create a pair of streams, one an `instream` and one an `outstream`, and associate them with a process. The string argument to `execute` is the operating-system command that starts the process, while the string list contains the command line arguments. The function `execute_in_env` is used to create a pair of streams, one an `instream` and one an `outstream`, and associate them with a process. The string argument to `execute_in_env` is the operating-system command that starts the process. The first string list contains the command line arguments. The second string list contains the environment. Each string has the form of a name, an "=", and a value, where name is the name of an environment variable.

The function `flush_out` ensures that the consumer associated with an `out_stream` has received all of the characters associated with that stream. It is provided primarily to allow the ML user to circumvent undesirable buffering characteristics that may arise in connection with terminals and other processes. All output streams are flushed when they are closed, and in many implementations an output stream is flushed whenever a newline character is written if that stream is connected to a terminal.

The function `can_input` returns the number of characters which may be read from its instream argument without blocking. For instance, a command processor may wish to test whether or not a user has typed ahead in order to avoid an unnecessary prompt. The exact definition of "currently available" is implementation specific, perhaps depending on such things as the processing mode of a terminal.

The `input_line` primitive returns a string consisting of the characters from an `instream` up through, and including, the next end of line character. If the end of stream is reached without reaching an end of line character, all remaining characters from the stream (*without* an end of line character) are returned.

Files may be opened for output while preserving their contents by using the `open_append` primitive. Subsequent `output` to the outstream returned by this primitive is appended to the contents of the specified file.

Basic support for the complexities of terminal I/O is provided. The pair of functions `is_term_in` and `is_term_out` test whether or not a stream is associated with a terminal; and `set_term_in` and `set_term_out` tell the ML system that a stream is (or is not) a terminal. These functions are especially useful with `std_in` and `std_out` because they are opened as part of the standard prelude. A terminal may be associated with a stream using the ordinary `open_in` and `open_out` functions; the naming convention to do this is implementation-dependent.

`use` and `use_stream` do nothing.

`exportML` creates a file containing the state of the world, called a heap file. The name of the file is given by the argument to `exportML`. The file is particular to the executable object file which created it; it cannot be used with any other executable object file. The executable object file may be restarted in the state it was in when the heap file was created by giving it the command line option `-s` *f*, where *f* is the name of heap file. This places the program in exactly the same state it was in when it wrote the file. For example, suppose we have created an executable object file `test` for the program:

```
structure S =
   struct
      val _ = (exportML "foo"; print "Hello\n");
   end
```

When `test` is run, it writes a file `foo`. If test is run again using the `-s` command line option:

```
test -s foo
```

it prints the following

```
Hello
```

`exportML` returns true when an executable object file is restarted using a heap file, and false when simply returning.

`exportFn` also creates a heap file that can be used to restart the executable object file which created it. The name of the heap file is given by the first argument. When the executable object file is restarted, it calls the function given as the second argument, then exits. The command-line arguments and environment are passed as the string list arguments to the function that is called. `exportFn` terminates execution of the ML system that called it.

5.4. Bool

```
signature BOOL =
  sig
    datatype bool = true | false
```

```
    (* stuck in here for convenience *)
    datatype 'a option = NONE | SOME of 'a

    val not : bool -> bool
    val print : bool -> unit
    val makestring : bool -> string
  end
```

The option datatype is one we have often found convenient. The definition of this structure is straightforward:

```
abstraction Bool: BOOL =
  struct
    datatype bool = true | false
    datatype 'a option = NONE | SOME of 'a
    fun not true = false | not false = true
    fun makestring true = "true" | makestring false = "false"
    fun print b = (output(std_out, makestring b); b)
  end
```

5.5. ByteArray

```
signature BYTEARRAY =
  sig
    infix 3 sub
    eqtype bytearray
    exception Subscript
    exception Range
    val array : int * int -> bytearray
    val sub : bytearray * int -> int
    val update : bytearray * int * int -> unit
    val length : bytearray -> int
    val extract : bytearray * int * int -> string
    val fold : ((int * 'b) -> 'b) -> bytearray -> 'b -> 'b
    val revfold : ((int * 'b) -> 'b) -> bytearray -> 'b -> 'b
    val app : (int -> 'a) -> bytearray -> unit
    val revapp : (int -> 'b) -> bytearray -> unit
  end
```

Byte arrays are just like arrays of integers, with the restriction that the values of the component integers must be between 0 and 255. The intent is that the implementation may store them more efficiently than the equivalent array.

Note that, by default, the ByteArray structure is present but not opened in the initial environment. The declaration `open ByteArray` may be used to open it. The use of ByteArray is

discouraged; future versions of the compiler may not support it, or (for example) debuggers might not support it.

The semantics can be defined by this implementation:

```
abstraction ByteArray : BYTEARRAY =
  struct
    infix 3 sub
    type bytearray = int array
    exception Subscript = Array.Subscript
    exception Range
    fun check x = if x<0 orelse x>255 then raise Range else ()
    fun array(i,x) = (check x; Array.array(i,x))
    val length = Array.length
    fun update(a,i,x) = (check x; Array.update(a,i,x))
    val op sub = Array.sub
    fun extract(b,i,0) = if i<0 orelse i>length(b)
                         then raise Subscript  else ""
      | extract(b,i,n) = chr(b sub i) ^ extract(b,i,n-1)
    val fold =  . . .
    val revfold =  . . .
    val app = ...
    val revapp = ...
  end
```

5.6. Integer

```
signature INTEGER =
  sig
    infix 7 * div mod quot rem
    infix 6 + -
    infix 4 > < >= <=
    exception Sum and Diff and Prod and Neg and Quot and Abs
    exception Div and Mod
    exception Overflow
    type int
    val ~ : int -> int
    val * : int * int -> int
    val div : int * int -> int
    val mod : int * int -> int
    val quot : int * int -> int
    val rem : int * int -> int
    val + : int * int -> int
    val - : int * int -> int
    val >  : int * int -> bool
    val >= : int * int -> bool
```

```
    val <  : int * int -> bool
    val <= : int * int -> bool
    val min : int * int -> int
    val max : int * int -> int
    val abs : int -> int
    val print : int -> unit
    val makestring : int -> string
  end
```

This should be mostly self-explanatory. The function `div` raises `Div` on divide by zero, otherwise `Overflow` if the result doesn't fit; similarly `mod` may raise `Div` or `Overflow`. Other operators may raise `Overflow` if the result does not fit into their representation. Standard ML of New Jersey uses finite precision signed 31-bit integers, which can represent a range from -2^{30} to $2^{30} - 1$.

The infix operators `quot` and `rem` implement rounding-towards-zero division and remainder; on nonnegative numbers they are equivalent to `div` and `mod`, but are always much faster. The `div` and `mod` required by the Definition disagree with division and modulus as implemented by typical computer hardware.

5.7. Real

```
signature REAL =
  sig
    infix 7 * /
    infix 6 + -
    infix 4 > < >= <=
    type real
    exception Sum and Diff and Prod
    exception Floor and Sqrt and Exp and Ln
    exception Div
    exception Overflow
    val ~ : real -> real
    val + : (real * real) -> real
    val - : (real * real) -> real
    val * : (real * real) -> real
    val / : (real * real) -> real
    val > : (real * real) -> bool
    val < : (real * real) -> bool
    val >= : (real * real) -> bool
    val <= : (real * real) -> bool
    val abs : real ->  real
    val real : int -> real
    val floor : real -> int
    val truncate : real -> int
    val ceiling : real -> int
    val sqrt : real -> real
```

```
    val sin : real -> real
    val cos : real -> real
    val arctan : real -> real
    val exp : real -> real
    val ln : real -> real
    val print : real -> unit
    val makestring : real -> string
  end
structure Real : REAL
```

This should be mostly self-explanatory. Except for the special exceptions `Floor`, `Sqrt`, `Exp`, `Ln`, raised by the functions of the corresponding names, all real-number functions raise only the `Real` exception with some system dependent argument string.

5.8. Ref

```
signature REF =
  sig
    infix 3 :=
    val ! : 'a ref -> 'a
    val := : 'a ref * 'a -> unit
    val inc : int ref -> unit
    val dec : int ref -> unit
  end
```

A reference is a cell which may be updated. The Standard ML of New Jersey compiler supports the applicative type variables found in [3]. It also supports *weak type variables*, which are a more flexible system for typing reference cells. The functions `inc` and `dec` can be defined as

```
  fun inc i = i := !i+1
  fun dec i = i := !i-1
```

5.9. String

```
signature STRING =
  sig
    infix 6 ^
    infix 4 > < >= <=
    type string
    exception Substring
    val length : string -> int
    val size : string -> int
    val substring : string * int * int -> string
    val explode : string -> string list
```

```
    val implode : string list -> string
    val <= : string * string -> bool
    val <  : string * string -> bool
    val >= : string * string -> bool
    val >  : string * string -> bool
    val ^  : string * string -> string
    exception Chr
    val chr : int -> string
    exception Ord
    val ord : string -> int
    val ordof : string * int -> int
    val print : string -> string
  end
```

Strings can be explained by the implementation below; of course, in practice a more efficient implementation is used.

```
abstraction String : STRING =
  struct
    infix 6 ^
    infix 4 > < >= <=
    type string = int list
    exception Substring
    val length = List.length
    val size = length
    fun substring(s,0,0) = nil
      | substring(a::b,0,len) = a::substring(b,0,len-1)
      | substring(nil,_,_) = raise Substring
      | substring(a::b,i,len) = substring(b,i-1,len)
    fun explode nil = nil
      | explode (i::l) = [i] :: explode l
    fun implode nil = nil
      | implode (s::l) = s @ implode l
    fun (_::_) > nil = true
      | nil > (_::_) = false
      | (i::r) > (j::s) = Integer.>(i,j) orelse i=j andalso r>s
    fun a <= b = not (a>b)
    fun a < b = b > a
    fun a >= b = b <= a
    val op ^ = op @
    exception Chr
    fun chr i = if i<0 orelse i>255 then raise Chr else [i]
    exception Ord
    fun ord nil = raise Ord | ord (i::r) = i
    fun ordof(s,i) = nth s handle Nth => raise Ord
    fun print s = (output(std_out,s); s)
  end
```

5.10. Bits

```
signature BITS =
  sig
    type int
    val orb : int * int -> int
    val andb : int * int -> int
    val xorb : int * int -> int
    val lshift : int * int -> int
    val rshift : int * int -> int
    val notb : int * int -> int
  end
structure Bits : BITS
```

The structure Bits allows shifting and masking of integers (viewed as strings of binary digits). This structure is present but *not* opened in the standard environment; its use is discouraged. The right shift (rshift) operator shifts sign bits into the left end of an integer.

5.11. System

```
signature SYSTEM =
  sig
    structure ByteArray : BYTEARRAY
    structure Control : CONTROL
    structure Tags : TAGS
    structure Timer : TIMER
    structure Stats : STATS
    structure Unsafe : UNSAFE
    structure Signals : SIGNALS
    structure Directory : DIRECTORY
    val exn_name : exn -> string
    val version : string
    val interactive : bool ref
    val system : string -> unit
    val argv : unit -> string list
    val environ : unit -> string list
  end
structure System : SYSTEM
```

Features of Standard ML of New Jersey that should not be expected in any other implementation of ML are grouped into the System structure, which is present but not opened in the standard environment.

The substructures `Control` of `System` are not documented.

The function `exn_name` returns the name of the exception constructor that was used to build a given exception value. The `version` string indicates which version of Standard ML of New Jersey is running. The `system` function runs an operating-system (shell) command specified by its argument. `argv` and `environment` return the command-line argument-list and (Unix) environment with which the Standard ML process was created.

6. Separate compilation

6.1. Description

Separate compilation provides support for the development of large programs using *sml2c*. Without separate compilation, one must recompile an entire program, even if only one file containing a small portion of the program is changed. This makes the development of large programs difficult, since recompilation of such programs takes a long time.

Separate compilation is based on the observation that large programs may be divided into files, each of which defines some modules. Each file may depend on other files, provided that the dependencies are non-circular. This allows programmers to avoid unnecessary recompilation, since when a file is changed only it and files which depend on it need to be recompiled.

All files may contain dependency information in them in the form of import statements. Each import statement lists separately compilable files which must be compiled before the file is compiled. Import statements have the form

`import` *"s1" ... "sn"*

where *"s1" ..."s2n"* is a list of one or more names. The names are used to find a source file and some other files according to the naming convention described in Section 6.2.

Each imported file is compiled in the original pervasive environment. The result environment is added to the environment of the file which imported the file.

Not all files can be separately compiled. *sml2c* (and SML/NJ) can separately compile only those files which contain signature and functor declarations. Files conatining structure declarations cannot be separately compiled.

Recompilation is avoided by saving the results of compilation. Corresponding to each source file, *sml2c* generates two files – one, a binary file, which contains environment information. and the other, an object file, which contains the executable code. The naming convention for these files is discussed in Section 6.2. When compiling a separately compilable file f, if the result files for f are more recent than the source file and files upon which f depends, the compiler obtains the the environment information it needs from the binary file and avoids compiling the source file.

6.2. Naming conventions

A naming convention is enforced for separately compilable files. A separately compilable file must have the `.sml` extension. An import statement of the form

```
import "a"
```

will import the file `a.sml`. The binary file name is generated by appending `.bin` to the base file name (the source file name without the `.sml` extension) and the object file name is generated by

appending `.o` to the base file name. For example, separate compilation of `a.sml` results in the generation of a binary file called `a.bin` and an object file called `a.o`

6.3. Suggested organization

To avoid unnecessary recompilation, we suggest that there be only one functor or signature declaration per file. Also, the files at the top-level (the ones whose names are given as arguments to *sml2c*), should be for "linking" only; they should only apply functors. Any other structures declared in those files will be compiled every time those files are compiled. For clarity, it is suggested that each functor should not refer to any other functors if possible; instead the file containing the functor should import only signatures. This will make each functor self-contained and easier to understand, since one needs to understand only what the argument structures matching the signatures are supposed to do.

6.4. Example

We have these files:

```
s1.sig.sml:
   signature S1 =
     sig
        type t
        val f : t -> t
     end

s2.sig.sml:
   import "s1.sig";
   signature S2 =
       sig
           structure S : S1
           val double : S.t -> S.t
       end

f1.sml:
   import "s1.sig";
   functor F1 () : S1 =
    struct
      type t = int
      val f = fn x => x+1
    end

f2.sml:
   import "s2.sig";
   functor F2(S : S1) : S2 =
       struct
```

```
          structure S = S
          val double = fn x => S.f (S.f x)
        end

link.sml:
   import "f1";
   import "f2";
   structure S1 = F1()
   structure S2 = F2(S1)
```

The following slightly edited terminal dialogue happened:

```
% sml2c -o test link.sml
  (* Separately compiled all of the .sml files; compiled link.sml.*)
% touch body.sml
% sml2c -o test link.sml
   (* Recompiled body.sml into body.bin, loaded all other binary
   files.  This illustrates the advantage of having files depend
   on body.sig.sml instead of body.sml.  More recompilation would
   have happened if source files depended on body.sml. *)
% touch body.sig.sml
% sml2c -o test link.sml
   (* Recompiled all of the .sml files; saving the results of
      everything but link.sml.  This illustrates the first point
      in the section on organization above, where a small
      modification can result in large amounts of recompilation.*)
```

References

[1] Andrew W. Appel and David B. MacQueen. A Standard ML Compiler. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274, pages 301–324. Springer-Verlag, 1987.

[2] Bruce F. Duba, Robert Harper, and David MacQueen. Typing First-Class Continuations in ML. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*. ACM, January 1991.

[3] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[4] J. Gregory Morrisett and Andrew Tolmach. A portable multiprocessor interface for standard ml of new jersey. Technical Report CMU-CS-92-155, School of Computer Science, Carnegie Mellon Unversity, June 1992. Also published as Department of Computer Science, Princeton University, Technical Report 376-92.

[5] John H. Reppy. Asynchronous Signals in Standard ML. Technical Report 90-1144, Department of Computer Science, Cornell University, August 1990.

[6] David Tarditi, Peter Lee, and Anurag Acharya. No Assembly Required: Compiling Standard ML to C. *Letters on Programming Languages and Systems*, 1992. To appear.