Debugging in Standard ML of New Jersey

AT&T Bell Laboratories
and Princeton University

September 17, 1992

1 Introduction

This is a reference manual for the Standard ML of New Jersey Debugger. It describes the system as of version 0.88. Changes from the previous version of the debugger (0.69) include mechanisms for interpolating code sequences during a debugging section, e.g., to change the value of a `ref` variable; replay support for programs using signals; more flexible breakpoint setting; improved dynamic type reconstruction; and numerous internal improvements. The Emacs interface has again changed slightly, so you will need to repeat the steps described in Section 3 to make the new `sml-debug.el` file accessible to Emacs.

Credit and Blame The debugger was designed and implemented by Andrew Tolmach. The Emacs interface was originally designed and implemented by Adam T. Dingle, who also wrote the original version of this manual. The original type reconstruction algorithm was created by David Tarditi.

Bug reports should be sent to `macqueen@research.att.com`.

2 The Debugger

The debugger is a specially-designed extension to the SML/NJ compiler. In its basic concepts and capabilities, the debugger resembles typical source-level debuggers such as `dbx`. You can set breakpoints, single-step execution, and display identifier values, while examining your source code in a separate window (managed by GNU Emacs). The debugger is implemented using a *source code instrumentation* technique. When directed to process programs for execution under debugger control, the compiler automatically inserts instrumentation code at all interesting *events*, including function applications, identifier bindings, the tops of function bodies, and so on. This instrumentation code allows the debugger to collect information and optionally gain control at these points in the program. Events are the units of single-stepping and setting breakpoints; thus, they play the role of line numbers in debuggers for conventional languages.

The debugger's most unusual feature is that it supports *reverse execution*; that is, it gives you the illusion of being able to execute your program backward as well as forward. This feature is implemented by taking periodic checkpoints of program state and logs of all I/O activities. To jump back to a previous point in the program's execution, the debugger restores the last known checkpoint prior to the target point, and then re-executes forwards. (Checkpoints consist of stored continuations, as captured by the `callcc` primitive, plus a description of the mutable `ref` and `array` data in the program.) To keep track of where it is in the program's execution, the debugger increments a counter each time an event is encountered. We refer to the value of this counter as the current *time*, and many commands take or return times.

In addition to being a valuable user-level feature, reverse execution is used internally by the debugger to support location-based breakpoints and identifier value lookup. The debugger's implementation is described in detail in Reference [1].

3 Setting Up

The SML/NJ debugger is intended to be run in GNU Emacs (or Epoch), allowing screen- and mouse-oriented "visual" debugging. Events are displayed within the source text itself; an ML program can be single-stepped, breakpoints can be set, identifiers' values can be displayed and the call traceback can be examined by typing key sequences in the source text window. The implementation extends Lars Bo Nielsen's Emacs mode for editing SML programs, which is now part of the GNU Emacs distribution.

To install, build a version of the compiler that includes the debugger if one does not already exist. See the instructions on building the compiler; include the `-debug` command line option to the `makeml` program. The executable image will be called `smld` by default. You will be able to use this image to compile code in debug mode or in the regular compilation mode; you may wish to hold onto the version of the compiler that does not support debugging since it is a few hundred kilobytes smaller. When compiling code outside debug mode, compilation and execution times should not differ noticeably between the two compiler images unless you are very tight on memory. After building the debugger, the `src/dbguser` directory should remain in place so that the debugger can find the special instrumented versions of library structures that are defined there (see Section 14.

Secondly, place the files `sml-mode.el`, `sml-init.el` and `sml-debug.el` (found in the `lib/emacs` directory of the SML distribution) in a directory where Emacs finds its Lisp code; you can find out what directories Emacs looks in by typing `C-h v load-path` in Emacs. Alternatively, you can place these three files in a directory of your own and tell Emacs to include that directory in its search path for Lisp code; to do so, enter the following line in your `.emacs` file (where `/u/me/myEmacsDir` is the directory where you would like Emacs to look):

```
(setq load-path (cons "/u/me/myEmacsDir" load-path))
```

Finally, add the following line to your .emacs file:

```
(load-library "sml-init")
```

`sml-init.el` includes code that sets up both sml-mode (the sml editing mode written by Lars Bo Neilsen) and sml-debug (debugging extensions to sml-mode).

You should be running GNU Emacs with a version number of at least 18.58. The system will probably work on earlier versions of Emacs as well, but has only been tested on 18.58. If you are running Epoch (a version of Emacs for the X Window System developed by Alan Carroll and Simon Kaplan at the University of Illinois at Urbana-Champaign), the debug mode will take advantage of Epoch's highlighting capabilities. You can obtain Epoch via anonymous ftp to cs.uiuc.edu. The debugger interface has been tested using Epoch 3.2-beta; it may not work properly with later versions of Epoch.

4 Getting Started

From within Emacs, type the command `M-x smld` to start up a window with the debugger version of SML running in it. This window is labeled `*Inferior smld*`. You use this window to start your program running, to provide keyboard input to the program, and to view output and final results. You can also use this window to issue typed commands to the debugger when you are stopped at a breakpoint (see Section 6); a special prompt `[dbg]-` indicates when the debugger is listening for such commands.

Begin by typing `usedbg "<filename>"` to compile and execute the contents of a file in passive debug mode. Typically, the file will contain function definitions. (In this version of the debugger, you must explicitly apply `usedbg` to every file to be compiled with debugging; nested `use`s will not work properly.)

Type `run "<declaration-or-expression>"` to compile and evaluate a given declaration or expression in active debug mode, which will allow you to step through program execution in a visual way. Typically, the argument to `run` will be an expression that applies one of the functions that was loaded using the `usedbg` command. (The argument should be given just as you would normally type it to the interactive system, except that double quotes (`"`) must be preceded by a backslash (`\`). No trailing semicolon is needed.)

In response to your `run` command, an Emacs window named `*sml-debug-command*` will be created containing the declaration or expression to be evaluated. The first *event* will be displayed in brackets in that window; it will look something like

```
[START:4]
```

where START indicates the type of the event and 4 is the current time. (For various technical reasons time does not begin at 1.)

Events are places in the ML program where you can stop execution to look at identifier values. You can use the Emacs key sequence `M-f` to *step* (execute forward one time step) until the next event is reached. (`M-f`, pronounced "meta-f", means that you should hold down the `META` key (often labeled `ALT`) and press the `s` key. If your keyboard does not have a `META` key, type `ESC` followed by `s`. For more information, see an Emacs reference manual, or type `C-h i` for information while in Emacs.) Try it. Try stepping a number of times; see what different types of events you reach.

As noted above, the SML/NJ debugger also permits you to step backward in time. The key sequence `M-b` steps backward one unit in time. Try stepping backward to the beginning of your program.

Sometimes you will want to "step over" a function call to stop at the next (or previous) event execution within the current function. You can skip forward in this way using `M-s` and backward using `M-r`.[1] You can also jump forward or backward to arbitrary times in the program's execution: type `M-<time> M-t` (that is, hold down the `META` key while you type the time number, then type `t` while still holding down `META`. If your keyboard does not have a META key, type `C-u <time> ESC t`.) Your jumps will always be truncated so that they lie between the beginning and ending times of program execution.

[1] The skip mechanism may not work reliably if your program contains `callcc`s and `throw`s.

5 Selecting Events

As your program executes, Emacs automatically displays the source file containing the current event. You can use the commands `M-n` and `M-p` to browse through events in your ML program. `M-n` moves the *event cursor* to the next event in the source code; `M-p` moves to the previous event. These commands do not cause any part of your program to be executed, or change the current time; they are simply a way of looking at the events that exist. The position of the event cursor is denoted by an event surrounded by square brackets; when you move the event cursor away from the current event, the current event remains visible, but is surrounded by a pair of angle brackets (`< >`) to distinguish it from the event cursor. Typing `M-c` will move the event cursor back to the current event.

Sometimes you may wish to move the event cursor to a point that is far away in your program text. To do so, move the Emacs cursor to the area of source code to which you wish to move the event cursor, then type the command `M-e`. The event cursor will jump to the event nearest to the Emacs cursor; you can then use the `M-n` and `M-p` commands to browse the events in that area. To move the event cursor to a different source file from the one in which it currently rests, visit the file using the ordinary Emacs command `M-x f`, and then use `M-e`.

If you are running Emacs under X, you can select events even more easily. Point to a character in your source code that is near the event that you would like to select, then hold down META and press the middle pointer button.

When two or more events are very closely related, so that the execution of one necessarily implies the immediate execution of the others, the debugger executes them all "at the same time." Such a set of events forms an *event cluster*. Any event in a cluster can be selected using the commands just described, but only the last event in a cluster is ever displayed as the current event when single-stepping, jumping, etc. For example, in the program

```
let [VAL] val a = 5
    [VAL] val b = 6
in f [APP] (a+b)
end
```

all three events belong to the same cluster and will appear to be executed simultaneously; only the `APP` event will ever be marked as current.

Section 13 describes all the possible event types and where they are displayed. Displaying event markers in the text at the correct spots is complicated by the presence of derived forms. In particular, curried functions turn into a nested chain of simple function definitions; the entry point to each function has a separate `[FN]` event, but all these events are displayed at the same location (after the `=` in the original `fun` declaration.

You may occasionally notice that a command causes the Emacs bell to ring and the message "No source available for [*event*:*time*]" to appear in the Emacs status window. This can occur for a number of reasons:

- If the *event* is `IO`, the debugger has halted within the standard IO library code. You can do single step backward using `M-b` to find out where the IO function was called from.
- If the *event* is `UNCAUGHT EXCEPTION`, you can type the `current()` command to display the exception; again, you can step backward with `M-b` to find out where the exception was raised.
- If the source file from which the code you are executing was compiled has since been recompiled, the debugger assumes that it is inappropriate to display locations in this file.

- Signal handler execution is bracketed with `ENTER SIGNAL HANDLER` and `EXIT SIGNAL HANDLER` events.

6 Breakpoints

To set a breakpoint at an event, move the event cursor to it and type the command `M-k`. The string `bk:` will appear inside the event, indicating that a breakpoint is set at it. The event will continue to be displayed even if you move the event cursor away from it. Type `M-k` again to remove the breakpoint at the event.

Use the `C-M-f` command to continue execution until the next breakpoint (or the end of your program) is reached. The `C-M-b` command moves backward in time until a breakpoint (or the beginning of your program) is reached.

A breakpoint can be set on any member of an event cluster, but it is always the last member of the cluster that is displayed as current when the breakpoint is reached.

You can also set breakpoints at particular times: type `M-<time> M-k` (that is, hold down the `META` key while you type the time number, then type `k` while still holding down `META`). Repeating the same command will delete the breakpoint you just created. Type `C-M-k` to get a list of time breakpoints.

The `M-- M-k` command deletes all existing breakpoints. (`M--` is "meta-hyphen".)

You can also set and view breakpoints by typing commands in the `*Inferior smld*` window. These commands use breakpoint *numbers*; each defined breakpoint gets a fresh number, which can later be used to identify or remove the breakpoint. The commands include:

```
- breakNear(<file>,<line number>,<line position>);
```

Place a breakpoint at the first event at or after the specified location. Return the breakpoint number as an `int option`, if any suitable event found.

```
- showEvents <filename>;
```

List all events defined for the specified filename; each event is identified by its location and type. This command is handy for constructing appropriate arguments to `breakNear`.

```
- breakWhen <time>;
```

Set a breakpoint at the time of the argument and return the breakpoint number.

```
- showBreaks();
```

Display a list of all breakpoints.

```
- deleteBreak <breakpoint number>;
```

Delete the breakpoint with the specified number.

```
- clearBreaks();
```

Delete all breakpoints.

You can associate an ML function with a breakpoint; the function is called whenever the breakpoint is reached. The function might print out the value of one or more identifiers. To associate a function with a breakpoint at a given event, move the event cursor to the event and type **bfunc <function>;** in the ***Inferior smld*** window. The function should have type **unit -> unit**.

For example, you might type

```
- bfunc (fn () => showVal "a");
```

in order to display the value of variable **a** (see Section 7) each time the breakpoint location is reached.

To reset the function associated with a breakpoint at an event, move the event cursor to the event and type `M-k M-k` (removing and reinstalling the breakpoint at the event).

To associate a function with a breakpoint at a given time, type

```
- tfunc <time> <function>;
```

To reset the function associated with a time breakpoint, type

```
- nofunc <time>;
```

7 Getting a call traceback and examining identifiers

The *backtrace cursor* is used to examine the chain of function calls that led to the point where execution is currently stopped. `M-u` moves the backtrace cursor, which initially rests on the current event, up one level in the function call chain. The backtrace cursor contains the string **:bt** inside it, and also displays the time at which the corresponding function call occurred. The command `M-d` moves the backtrace cursor down one level in the function call chain. As a convenience, typing `M-t` while a backtrace event is selected will jump to the time of the backtrace event. You can also use the typed command

```
- showCalls <frames>;
```

which produces a formatted display of the specified number of call frames, including the values of the function parameters bound at each call.

The debugger's *current scope* is the static scope of the program at the backtrace cursor (again, this is initially the current event, i.e., the event where execution last stopped). To display the value of any identifier in the current scope, simply type it at the **[dbg]-** prompt in the ***Inferior smld*** window. In fact, you can type any arbitrary ML expression; any identifiers mentioned will be bound to their values in the current scope. Note that you can use expression evaluation to pick apart complex values, e.g, to subscript into arrays. You may also invoke your own arbitrary ML functions (e.g., data structure pretty-printers) if they are defined in the current scope (i.e., in the program being debugged prior to the backtrace time or at top level before you started this debugging session).

Expression evaluation of this sort never changes the behavior or state of the program being debugged; time "stands still" during the evaluation and any location breakpoints in functions being executed are ignored. Also, although you can execute **ref** and array assignments, the effect of these assignments will be lost as soon as you resume program execution (see Section 9 to find out how to make assignments stick). You can, however, declare new values and functions in the usual manner at any time. Such "hand-entered" declarations (actually, you can `use` them from a file) remain visible when you execute to another point in the program, and even after the program completes. They can be very handy for temporarily holding values and specialized debugging functions. These declarations hide any program declarations using the same identifier name, and the hiding effect can't be removed, so be careful!

Sometimes it is useful to see the spot in your program where an identifier was bound. To do this, type `M-l` and enter the identifier's name in the Emacs minibuffer. The identifier's value will be printed out, and the event cursor will move to the event at which the identifier's value was bound. This is particularly handy if the identifier represents a function. If you are running Emacs under X, you can find an identifier's bounding location by pointing to its name, holding down META and pressing the first mouse button.

You can also use the typed command

```
- showVal <identifier name>;
```

to display the value,type, and binding location of the specified identifier. It is normally easier just to type the variable at the `[dbg]-` prompt, but `showVal` can be handy in constructing macros or for the `bfunc` command (see Section 6).

8 Interrupting Execution

You can halt your program at any time during forward execution (e.g., if it enters an infinite loop) by typing the `C-c C-c` command in the `*Inferior smld*` window. The program will always halt on an event boundary, and the location and time should be shown in source code window. You can then use the traceback commands to find out where you are in the execution.

9 Executing Interpolations

Interpolations are code sequences executed "in the middle of" the ordinary user program, after stopping at some event under debugger control. Unlike ordinary expression evaluation in the debug monitor, interpolations can alter the state of the user program, and they become a permanent part of the program's execution history. Their main use is to update mutable variables (e.g., by executing assignments), but they may contain arbitrary code sequences. Interpolations are parsed and executed in the environment corresponding to the event time at which they were entered; thus, they can reference and side-effect variables in scope at that time. They can also perform other side-effects, such as raising exceptions (which might be useful to test the program's handler code), perform I/O, etc.

An interpolation is entered by typing `execute "<declaration or expression>"`. (As with the `run` command, the argument should be given as you would normally type to the interactive system, except that double quotes (`"`) must be prceded by a backslash (`\`).) The argument string will be compiled and and evaluated immediately; evaluation will typically involve the execution of several events, and will advance the current time by several steps. At a minimum, each interpolation has an associated START and END event, similar to those for the entire program unit. Execution will halt at the END event to allow further debugger commands to be entered. All events executed during the interpolation become part of the program's history, and can be revisited using the normal time-travel commands, although this is not ordinarily useful. Since the interpolation may change the course of program execution, any existing program history for later times is erased when the interpolation is performed. Whenever the part of the program containing the interpolation time is re-executed, the interpolation itself will be re-executed at the correct point; on re-execution, however, execution does not halt at the interpolation's END event, but continues to the next event in the ordinary program.

10 Finishing Up

Two commands are available to exit from an active debugging session. The `C-c C-d` command aborts execution of the program run you started with the `run` command; no changes are made to the top-level environment. The `C-M-c` command completes execution of the program run; if the argument you gave to `run` included declarations, they are added to the top-level environment. Both commands remove all event labels from source code windows and delete the `*sml-debug-command*` buffer. The `*Inferior smld*` window will remain, so that you can load new code or specify another command to run.

While source code windows have events displayed in them, they are placed into Label mode, a read-only Emacs minor mode. This prevents you from editing them and unintentionally saving event labels into your source files. Before you edit a section of source code that is being debugged, you should use the `C-c C-d` command to abort the debugging session and remove the labels. If for some reason a window remains in Label mode after you have exited the debugging session (this should not happen) you can use the command `C-x C-q` (exit-label-mode) to remove the labels from the window.

11 Running without Emacs

You can run the `smld` executable directly, just like ordinary `sml`. Only the typed commands (summarized in Section 12) may be used. The only difference in their behavior has to do with specifying and reporting source-code locations: file names are taken relative to the current working directory as in normal SML/NJ (rather than being expanded to absolute names under the Emacs-based debugger) and file positions are reported as line number and position (rather than as absolute character position).

12 Command and Function Summary

The following commands are available within the `*Inferior smld*` process as well as within windows containing source code. `<left>` and `<middle>` refer to the left and middle pointer buttons when running under the X window system.

Emacs Command	Default Key Binding	Description
sml-step	M-f	single-step forward
sml-step-backward	M-b	single-step backward
sml-skip	M-s	skip forward
sml-skip-backward	M-r	skip backward
sml-goto-time	M-<time> M-t	go to specified time
sml-select-next	M-n	select next event
sml-select-previous	M-p	select previous event
sml-select-current	M-c	move event cursor to current event
sml-select-near	M-e	select event near cursor (source windows only)
sml-mouse-select-near	M-<middle>	select event near pointer (source windows only)
sml-break	M-k	toggle breakpoint at selected event
sml-break	M-<time> M-k	toggle breakpoint at given time
sml-show-breaktimes	C-M-k	list breakpoint times
sml-break	M-- M-k	delete all breakpoints
sml-proceed-forward	C-M-f	execute forward to next breakpoint
sml-proceed-backward	C-M-b	execute backward to previous breakpoint
sml-up-call-chain	M-u	move backtrace cursor up the call chain
sml-down-call-chain	M-d	move backtrace cursor down the call chain
sml-goto-time	M-t	go to time of selected backtrace event
sml-variable-value	M-l or M-<left>	show variable value and binding event
sml-abort	C-c C-d	abort program execution and exit debugging session
sml-complete	C-M-c	complete program execution and exit debugging session

The following ML functions can be typed in the ***Inferior smld*** window, and can also be used when running the debugger without Emacs.

```
val usedbg: string -> unit
```

Compile and execute a file in passive debug mode, so that compiled code will be instrumented for debugging. All files used by the file will also be executed in passive debug mode.

```
val usedbg_string: string -> unit
val usedbg_stream: instream -> unit
```

Compile and execute the text of a string or an arbitrary input stream in passive debug mode.

```
val usedbg_live: string -> unit
```

Compile and execute a file in active debug mode. For a typical file containing definitions of functions, structures, and functors, this is chiefly useful as a way to stepping through functor applications.

```
val usedbg_script: instream -> instream -> unit
```

Compile and execute contents of first instream in active debug mode, taking debug commands from second instream. Useful for testing.

```
val cd: string -> unit
```

Change the directory for subsequent relative file names.

```
val run: string -> unit
```

Compile and execute a command in active debug mode.

```
val execute: string -> unit
```

Compile and execute an interpolation; see Section 9.

```
val breakNear: string * int * int -> int option
val showEvents: string -> unit
val breakWhen: int -> int
val showBreaks: unit -> unit
val deleteBreak: int -> unit
val clearBreaks: unit -> unit
val bfunc: (unit -> unit) -> unit
val tfunc: int -> (unit -> unit) -> unit
val nofunc: int -> unit
```

Breakpoint control commands, described in Section 6.

```
val current: unit -> unit
```

Display information about the current event, including any uncaught exception or pending interrupt.

```
val showCalls : int -> unit
```

Display specified number of frames of the call stack; see Section 7.

```
val showVal : string -> unit
```

Display value and binding location for specified variable; see Section 7.

```
val forward: unit -> unit
val backward: unit -> unit
val jump: int -> unit
```

Move forward or backward to the nearest breakpoint, or jump to specified time.

```
C-c C-c
```

Halt forward execution of the current program.

```
val complete: unit -> unit
val abort: unit -> unit
```

Finish debugging program, adding any declarations to the top-level environment (`complete`) or discarding them (`abort`); see Section 10.

```
val haltOnSignal: signal * bool -> unit
val setSignal: signal -> unit
val clearSignal: signal -> unit
```

These commands are described in Appendex A.

13 Event Types

This section describes the event types may be displayed inside the event cursor; an example is given for the common event types. The most common events are:

- `START` Start of program execution or interpolation.
- `APP` Application event. Occurs just before a function is applied to its argument; appears between the function and its argument. When an infixed operator is applied, the event appears immediately following the operator.

  ```
  let val a = myfun [APP] 7 in ...

  infix myOp
  let val b = 2 myOp [APP] 8 in ...
  ```

- `VAL` Value binding event. Occurs just after a value has been bound.

  ```
  let [VAL] val a = f(7) in ...
  ```

- `VALREC` Recursive value binding event. Analogous to `VAL`.

  ```
  let [VALREC] fun f x = if x = 0 then 1 else x * (f(x-1)) in ...
  ```

- `FN` Function entry event. Occurs just after a function has been entered. If a function is defined by multiple rules, there is a different `FN` event for each. If the function is curried, there will be separate `FN` events for each argument, all (confusingly) displayed at the same source code location (after the `=`). Extra `FN` and `RAISE` events (see below) are generated for the implicit branch generated by the compiler in case a pattern match fails; these events are displayed immediately after the last explicit clause but cannot be discovered by `M-n`, `M-p`, or `M-e`.

  ```
  fun f (SOME x) = [FN] g x
    | f NONE = [FN] 0
  ```

- `CASE` Case entry event; analogous to FN.

```
case d of SOME x => [CASE] g x
        | NONE => [CASE] 0
```

- `RAISE` Exception raising event. Occurs just before an explicit exception is raised. See note on implicit Match exceptions in description of `FN` events, above.

```
case v of SOME x => g x
        | NONE => raise [RAISE] MyException
```

- `HANDLE` Exception handling event. Occurs when an exception is handled, before the exception handling code is actually executed.

```
fun myHd x = SOME (hd x) handle Hd [HANDLE] => NONE
```

- `LET` Occurs after a set of `let`-bound declarations but before the expression body has been evaluated; i.e., at point where the `in` keyword appears in the program text.

```
let val a = 10 [LET] in a + 2 end
```

- `LOCAL` Occurs before a set of local declarations are evaluated.
- `LOCAL IN` Occurs after a set a local declarations are evaluated, but before the set of declarations which contain the local declarations in their score are evaluated, i.e., at the point where the `in` keyword appears in the program text.
- `LOCAL END` Occurs at the end of a local block.

```
local [LOCAL] val a = 10 in [LOCAL IN] val b = a [LOCAL END] end
```

- `END` End of program execution or interpolation.

The following events are not associated with a particular location in your source text:

- `IO` Occurs just before certain operations in the standard IO library.
- `UNCAUGHT EXCEPTION` Occurs immediately after an uncaught exception is raised.
- `ENTER SIGNAL HANDLER` Occurs immediately before executing a user signal handler.
- `EXIT SIGNAL HANDLER` Occurs immediately after executing a user signal handler.

The remaining types of events are encountered less often, since code that contains these events is usually not executed in active debug mode.

- `STRUCTURE` Structure declaration event.
- `ABSTRACTION` Analagous to `STRUCTURE`.
- `FUNCTOR` Functor declaration event.
- `SIGNATURE` Signature declaration event.

- `FUNCTOR ENTRY` Occurs upon entry to a functor.
- `FUNCTOR APP` Occurs just before a functor is applied to its argument.
- `STRUCTURE END` Occurs at the end of the declarations in a structure.
- `STRUCTURE VAR` Occurs just before a structure variable is used.
- `OPEN` Occurs just before a structure is opened.
- `TYPE` Type declaration event.
- `FIXITY` Fixity declaration event.
- `EXCEPTION` Exception declaration event.

14 Environmental Issues

"Debuggable" code (i.e., code compiled via `usedbg`, `run`, etc.) uses a special version of the pervasive environment. In particular, it redefines:

- the `Ref` and `Array` structures to support checkpointing of the mutable store;
- the `IO` structure to support logging of IO events;
- the `System.Signals` structure to support logging of signals;
- various other structures that use `Ref`, `Array`, or `IO`; and
- the `List` and `General` structures, for reasons explained below.

These redefinitions should be transparent to you, unless, of course, you rebind these structures yourself. (Hint: Don't). The debugger also defines top-level structure UserDebugCommands, which contains the commands described in this manual; don't redefine this structure!

The debugger copes adequately on code that performs its own `callcc`s, except that `skip()` and `skip_backward()` are unreliable.

It is possible to mix non-debuggable with debuggable code under the following (rather heavy) restrictions:

1. The non-debuggable code must not update any arrays or `ref`s created by debuggable code, and must not create arrays or `ref`s which are then passed to debuggable code. (N.B. There are many ways to pass an object, including: as an argument to a function, as the value of a `ref` or array, or as the argument to an exception constructor.)
2. The non-debuggable code should not apply debuggable functions.

Special pre-instrumented version of the `List` and `General` structures are provided because they would otherwise violate restriction 2; execution within functions in these structure can be traced in the normal manner, provided the `dbguser` directory is left in place after the debugger is built.

15 Performance

Because it expands the effective size of the source code by instrumentation, the debugger can put heavy demands on system resources. Compilation times, in particular, will suffer a many-fold increase, particularly in systems where memory is tight. One partial solution is to use interpreted execution mode where possible. The interpreter is enabled by typing

```
- System.Control.interp := true;
```

at top level.

The I/O and reference memory logs can also occupy a great deal of memory. In particular, remember that every byte of input read (from a terminal or a file, in the present implementation) is stored in the I/O log and becomes a permanent part of the program's live data.

Execution times for debuggable code should normally be 2-4 times slower than ordinary code. Heavy use of I/O and mutable store (particularly the creation of references and arrays) may lead to poorer performance.

16 Known Problems

- Debugging of programs with byte arrays or real arrays is not supported.
- The `M-e` key command doesn't work reliably.
- Superfluous messages can appear in the `*Inferior smld*` window when executing key commands in rapid succession.
- Type-ahead to the `*Inferior smld*` window is usually lost.
- The types of identifiers within functor bodies that depend on types in the functor parameter are not computed correctly.
- Some substructures are not properly constrained by signatures, so you can get values that should be hidden.
- Certain private structures containing debugger control support functions and debugger versions of the pervasives are visible in `smld`.
- Epoch support needs to be brought up to date.

17 The Future

In addition to fixing some of the problems mentioned in this document (as well as the problems we don't know about yet) we plan the following major extensions to the debugger's functionality:

- There will be proper support for integrating debuggable code with non-debuggable code in a single execution.
- We are investigating integration with the separate compilation mechanism.
- We'll support more efficient logging of input from files, and more efficient logging of the mutable store using a new multi-generational garbage collector.

A Signals

This appendix describes advanced features for debugging user programs that perform explicit signal handling. The debugger supports signal handling by logging signals during recording phases of execution and simulating them during replay phases. Instrumented code is compiled using a special debugger version of the signal primitives (as defined in System.Signals) to obtain debugger support. The debugger supports user programs handling all signal types supported by SML/NJ except `SIGINT`, which is reserved for interrupting the debugger itself (see Section 8).

There are two stages to signal processing under debugger control. In the first stage, signals are recorded in a *pending* vector. This vector is updated when the debugger is in recording mode (i.e., executing forward through a previously unexplored region in time) and receives an asynchronous signal in the usual way (e.g., from a timer or other external agent). In the second stage, a pending signal is *delivered* to the user's signal handler. Delivery of signals is synchronized with debugger events: deliveries always occur just after a debugger event has been executed, so they can be given precise time stamps. Delivered signals are logged and re-delivered at the same time stamp during re-execution.

You can alter normal signal handling behavior at several points. Entries in the pending vector can be explicitly set and cleared whenever the debugger is halted. You can also make the debugger halt and display a message prior to delivery of specified signal types; if the relevant entry in the pending vector is cleared at this point, the signal will not be delivered.

Pending Vector Signals are normally entered in the pending vector by the debugger's underlying signal handler. Signals are recorded in the vector only when the debugger is in recording mode; at other times the debugger's handler simply ignores them. Note that a signal might be received even during very brief recording phases (e.g., single-stepping) but is much more likely to occur during prolonged forward execution. At most one signal of any type can be pending at any given time; receipt of additional signals only increases the "count" of the signal ultimately delivered.

The debugger always handles and records signals for which user handlers have been installed. In addition, it handles and records signals which would normally cause SML/NJ to terminate if not handled, i.e., `SIGHUP`, `SIGALRM`, `SIGTERM`, `SIGUSR1`, `SIGUSR2`, and `SIGVTALRM`. (`SIGINT` is handled separately by the debugger and cannot be handled by your program.) Other signal types are not handled by the debugger unless you have installed a signal handler for them, so they are given SML/NJ's default treatment. In particular, `SIGQUIT` terminates the entire `smld` execution; `SIGURG`, `SIGCHLD`,`SIGIO`, `SIGWINCH`, and `SIGGC` are ignored.

Any signal can be explicitly set in the pending vector by the command

```
- setSignal <signal>;
```

which increments the pending count for the specified signal type. The specified type need not have a user handler installed. The command

```
- clearSignal <signal>;
```

zeroes the pending count for the specified signal type.

The pending signal vector is only relevant to execution in recording mode. The entire vector, including any explicit user alterations, is automatically cleared by any explicit reverse motion command (e.g., jump to an earlier time). The `setSignal` command can still be used after such a reverse motion; in addition to setting a pending vector entry, it erases any subsequent execution history, including signal deliveries, I/O, interpolations, etc., to be discarded. Thus, any subsequent forward execution will place the debugger in recording mode, so the specified signal will be delivered immediately. The `clearSignal` command can also be used, but its only effect is to erase execution history, thus in particular preventing re-delivery of the specified signal (and any others!). There is no mechanism for selectively editing the log of delivered signals.

Signal Delivery Signals are delivered during recording phase at the first possible event time following their entry in the pending vector. Usually this will be the first event following their entry, but delivery is delayed until (a) signals are not masked, (b) no signal handler is currently being executed, and (c) no interpolation is executing or has been set. Delivery of a signal clears it from the pending signal vector and enters it in the log. On replay, the signal is re-delivered at the same event time.

At most one signal is delivered at any given event time. If multiple signals are pending, they are delivered in the numeric order of their codes as described in `runtime/ml_signal.h`; each signal after the first is delayed until the previous signal's handler completes.

Any signal type lacking a user-installed handler is given an implicit nop handler, so there is no harm in delivering it.

Halting on Signals By default, the debugger delivers signals to installed user handlers quietly, but halts before delivering signals for which no user handler is installed, displaying the name of the signal. The user command

```
- haltOnSignal(<signal>,<halt>:bool);
```

can be used to override this default action and make the debugger always either stop or not stop before delivering signals of a specified type.

Signal Handler Execution Signal handlers are executed under debugger control just like normal code. Execution of the handler is preceded by a `ENTER SIGNAL HANDLER` pseudo-event and followed by a `EXIT SIGNAL HANDLER` pseudo-event. It should be possible to study various forms of handler error, such as throwing to a continuation rather than returning one. Uncaught exceptions raised in a signal handler, which normally cause SML/NJ to terminate, produce an Uncaught Exception condition just as in an ordinary program body.

References

[1] A. P. Tolmach, *Debugging Standard ML*, Ph.D. Thesis, Princeton University, July 1992. Available as Princeton Univ. Dept. of Computer Science Tech. Report 378-92.