Debugging in Standard ML of New Jersey

Andrew P. Tolmach and Adam T. Dingle
Princeton University

March 17, 2015

1 Introduction

This is a (still) *preliminary* reference manual for the Standard ML of New Jersey Debugger, designed and implemented by Andrew P. Tolmach with an Emacs interface by Adam T. Dingle.

This document describes the system as of version 0.69. Changes from the original system include the ability to evaluate arbitrary expressions, a new notion of identifier scope, commands for "skipping" forward and backward, and revised key bindings. There are some additional new features that will be documented when they settle down and become more presentable. Note that because the Emacs interface has changed somewhat, you will need to repeat the steps described in 3 to make the new sml-debug.el file accessible to Emacs.

Further changes are likely. Your comments and suggestions are welcomed. Bug reports are also welcome, but no guarantees are offered as to when they will be dealt with.

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. Single-stepping units and breakpoints are defined in terms of events; in this sense, 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. More details on the implementation philosophy may be found in [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.

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. You can alternatively 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 18.55. The system will probably work on earlier versions of Emacs as well, but has only been tested on 18.55. 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.

4 Getting Started

From within Emacs, type the command `M-x sml` to start up a window with SML running in it. In that window, type `usedbg "<filename>"` to compile and execute the contents of a file in passive debug mode. Typically, the file will contain function definitions. (When you `usedbg` a file, all files `used` by that file will be compiled with debugging as well. The semantics of `use` are tricky, even

when the debugger is not involved; it is strongly recommended that any file containing one or more `use`s contain *only* `use`s.)

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`. 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.

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.

5 Selecting Events

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, or in a different source file than the one in which the event cursor currently rests. 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.

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.

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.

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 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 `breakFunc <function>;` in the SML process window. The function should have type `unit -> unit`.

For example, you might type

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

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

```
- breakTimeFunc <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.

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 prompt in the debugger process 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. 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. You may notice that if you move the event cursor away from the event and then try to return to it, you might not be able to find the event again; this is because not all events are normally displayed (see the introduction to section 11 for details).

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.

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 window containing the sml process. 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 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 window containing the SML process 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.

10 Command and Function Summary

The following commands are available within the window containing the sml 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.

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 useful ML functions are available in the sml process window:

`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.

`usedbg_stream: instream -> unit`

Compile and execute the text of an input stream in passive debug mode.

`uselive: 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.

`run: string -> unit`

Compile and execute a command in active debug mode.

`bfunc: (unit -> unit) -> unit`

Sets the break function for the breakpoint at the selected event. (To reset the break function, select the event and type `M-k` twice.)

`tfunc: int -> (unit -> unit) -> unit`

Sets the break function for the breakpoint at a given time.

`nofunc: int -> unit`

Resets the break function for the breakpoint at a given time.

`C-c C-c`

Halt forward execution of the current program.

11 Event Types

This section describes the event types may be displayed inside the event cursor; an example is given for the common event types.

Note that, although all these types of events are produced internally, they are not all visible to `M-n`, `M-p`, and `M-e`. When a set of several events is very closely related (generally when the execution of one necessarily implies the execution of the others), only the last event (in execution order) is normally visible. (The others may appear as binding sites for identifiers when `M-l` is used.) However, enough events will always be visible to allow you to set a breakpoint to examine any identifier value of interest. For example, in the code

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

only the APP event will be visible, and you can examine the value of `a` or `b` by breaking there; you cannot break at either VAL event.

A further note: displaying event markers in the text at the correct spots is complicated by the presence of derived forms; we try to do the best we can in these cases, but there may still be some confusions.

The most common events are as follows: `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 a = f(7) [VAL] in ...
```

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

```
let val rec f = fn x => if x = 0 then 1 else x * (f(x-1)) [VALREC] in ...
```

`FN`: Function entry event. Occurs just after a function has been entered. If a function's argument consists of several different patterns, there will be a FN event for each. Note that an extra FN event (and RAISE event; see below) are generated for the implicit branch generated by the compiler in case a pattern match fails.

```
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 exception is raised. See also the comment about 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
```

`START`: Start of program execution.

`END`: End of program execution.

The following types of events are normally not visible:

`LET`: LET body event. Occurs after a set of LET-bound declarations but before the expression body has been evaluated; in other words at the point where the `IN` keyword appears in the program text.

`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. In other words, occurs at the point where the `IN` keyword appears in the program text.

`LOCAL END`: Occurs at the end of a local block.

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.

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 binding event./

`ABSTRACTION`: Analagous to STRUCTURE.

`FUNCTOR`: Functor binding event.

`SIGNATURE`: Signature binding 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.

12 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;
- various other structures that use REF, ARRAY, or IO; and
- the LIST structure, 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 makes and uses top-level definitions of two structures named DEBUGN and DEBUGZ, so your code shouldn't redefine these at top level.

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 refs created by debuggable code, and must not create arrays or refs 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.

A special pre-instrumented version of the LIST structure is provided because it would otherwise violate restriction 2; execution within function in this structure can be traced in the normal manner.

13 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 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.

14 Known Problems

- The extra FN and RAISE events mentioned in section 11 are very irritating and confusing, especially since the FN event is visually indistinguishable from the last legitimate FN event for the declaration.

- There needs to be a convenient way of setting breakpoints at value binding events, not just at the events visible when single-stepping.

- The `M-e` key command doesn't work reliably.

- Superfluous messages can appear in the debugger process window when executing key commands in rapid succession.

- The types of some identifiers within functor bodies are not computed correctly.

15 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:

- It will be possible to change the values of ref and array variables, by evaluating assignment expressions. (Support for this is already present in an experimental form.)
- 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.

References

[1] A.P.Tolmach, and A.W.Appel, "Debugging Standard ML Without Reverse Engineering," *Proc. 1990 ACM Conference on Lisp and Functional Programming*, June, 1990.