Psd – a Portable Scheme Debugger*

Pertti Kellomäki, pk@cs.tut.fi
Tampere University of Technology
Software Systems Lab
Finland

August 19, 1997

Abstract

Psd is a portable debugger for the Scheme language. Debugging with Psd is accomplished by instrumenting the original source program. When the instrumented program is run, it presents the user with an interactive interface that lets him or her examine and change values of variables, set breakpoints, and single step evaluation. Psd is designed to be run within GNU Emacs, which is used for displaying the current source code position.

1 Introduction

There are numerous implementations of the Scheme language available. While some of them have extensive debugging capabilities, many small implementations have only limited support for it. Psd provides source level debugging as an "add on", relying only on features described in the Revised^4 Report on the Algorithmic Language Scheme [3]. Psd does not use the macro proposal of the report, so it should work with Revised^3 Scheme implementations, also.

Psd works by transforming the original program into an operationally equivalent program (modulo the debugging capabilities), that allows the user

* To appear in Lisp Pointers

to examine and change variables, set breakpoints, and single step the evaluation process. To a calling procedure, a debugged procedure behaves exactly like the original. This allows mixing debugged and non-debugged code.

When a program is debugged with Psd, its source code is first given to the instrumenting part of Psd. The instrumentation part writes an instrumented version of the program to a file, which is then loaded into the Scheme environment. When a procedure in the program is invoked, it behaves as if it was executed under a conventional debugger.

2 Related Work

There are a few other debugger that are implemented similarily. The edebug package for GNU Emacs Lisp [2], written by Daniel LaLiberte, uses the the same ideas, but is much more tightly integrated with GNU Emacs. Jurgen Heymann has implemented an instrumenting debugger for the Simscript II.5 simulation language [1].

3 The Emacs Interface

Psd uses GNU Emacs as its user interface. The primary use for Emacs is to provide source code debugging. A typical Psd session is shown in figure 1.

Psd uses the same interface to Emacs as the GNU project debugger Gdb, and the Psd interface was actually modified from the existing Gdb interface. When an instrumented program is run, it emits specially formatted lines containing the source file and line number of the current source line. Emacs interprets these lines by showing the appropriate file in an editing window with an arrow indicating the current line.

There is a small amount of Emacs Lisp code that interacts with the Scheme environment. Emacs generates temporary file names and issues instrumenting and loading commands. The instrumenting code is file oriented, but with the Emacs interface it is possible to pick one procedure from a source file to be debugged. The Emacs interface is also used for setting breakpoints, with the Emacs Lisp code taking care of the low level details like file names and line numbers.

Figure 1: A Psd Session

```
(let ((x 1))
  (+ x 1))

(let ((x 1))
  (let ((psd-val
         (lambda (sym)
           (case sym
             ((x) x)
             (else (psd-val sym))))))
  (psd-debug psd-val (lambda () (+ x 1)))))
```

Figure 2: Accessing variables by name

4 Accessing Variables by Name

One of the main uses of a debugger is examining the values of variables. In some Lisp environments it is easy to provide access to variables by starting a new read-eval-print loop. The Scheme report does not include `eval`, however, so a different strategy must be used. In Psd this problem is solved by inserting an access procedure each time new variable bindings are made. This procedure performs the mapping between symbols and actual program variables. Figure 2 shows a `let` form and the code that Psd generates for it.

The procedure `psd-val` is passed to the debugger command loop `psd-debug`. Using it the command loop gets access to variables in the current lexical environment of the debugged program. The scope rules come "for free", because the name `psd-val` in the body of `psd-val` refers to the lexical environment surrounding the `let` form.

Assignments to local variables are made using the same mechanism. A setter procedure `psd-set!` is inserted each time local variables are defined, and it is also passed to `psd-debug`.

Access to global variables is provided using the same idea. Every instrumented Scheme file includes definitions for procedures similar to `psd-val` and `psd-set!`. When the file is loaded, the procedures are added to a global access procedure list. The global definitions of `psd-val` and `psd-set!` call the access procedures one by one until either the access succeeds or there are no more access procedures.

The Psd runtime support includes access to all the essential procedures* described in the Revised^4 Report. The debugger command loop includes a simple evaluator that can evaluate calls of the procedures that are visible to it. The lack of a `bound?` predicate or some other portable way of finding out whether an identifier is bound prevents access to the non-essential names in the report.

5 Breakpoints and Single Stepping

In order to be able to single step the evaluation process, the debugger must be able to gain control both before and after each expression is evaluated. In Psd this is accomplished by packaging each expression inside a procedure. This procedure is then passed to the debugger command loop. When the user wants to continue, the command loop simply calls the procedure that was passed to it. The command loop then gains control again, and finally returns the value that the procedure returned. For example, the expression `(+ x 1)` in figure 2 is transformed to

```
(psd-debug (lambda () (+ x 1)))
```

The transformation is done recursively, so the expression is really transformed into

```
(psd-debug (lambda () ((psd-debug (lambda () +)
                       (psd-debug (lambda () x)
                       (psd-debug (lambda () 1)))))))
```

In reality the debugger gets some more information (the current location in source code etc.), but the basic idea is the same.

It may seem that there is no point in instrumenting expressions like `+` and `1`, but the instrumentation is needed for supporting breakpoints. Breakpoints are implemented by maintaining a list of source code locations of breakpoints. Each time `psd-debug` is called, it checks if there is a breakpoint for the current source line, and starts a command loop if needed. If

* The report distinguishes between essential procedures that a conforming implementation must provide, and non-essential procedures that are not required.

primitive expressions like `x` would not be instrumented, there would be source lines for which breakpoints could not be set, for example in

```
(foo bar
     baz
     zap)
```

Single stepping is implemented similarily. Stepping by line is implemented by keeping track of the line number corresponding to the previous call to `psd-debug`.

6 Runtime Support

Psd needs some runtime support in the Scheme environment. The command loop `psd-debug` is a closure containing state variables for the debugger. Procedure application needs the procedure `psd-apply`, and breakpoint support needs a global variable for storing the breakpoint locations.

The instrumentation code resides in the same Scheme environment as the debugged program. This is not strictly necessary, but it has proved to be a convenient way of working. For example, it is easy to cut down the size of the instrumented files by assigning a unique integer for each source file name and using it instead of the full path name.

7 Catching Runtime Errors

A typical use of a debugger is to let the program run until a runtime error occurs and examine the program state to find out what went wrong. With Psd, real runtime errors can not be allowed to happen, since it relies on correct execution of the instrumented code. Instead, if an expression would cause a runtime error to occur, the command loop is called and an error message is issued.

Aside from syntactically incorrect expressions and causes outside the scope of the language (exhaustion of memory, receiving a signal etc.), the only place where a runtime error can occur is the procedure call. When calling a user defined procedure, the only possible error is that a wrong number of arguments is supplied. Although it would be possible to detect at least some of these errors, Psd does not currently check the number of arguments to a user procedure.

The number of primitive procedures is fixed, so they are easier to handle. Psd transforms each procedure call `(proc args)` into `(psd-apply proc args)`. Before `psd-apply` applies the procedure to its arguments, it checks if the procedure is a primitive procedure. If it is, `psd-apply` checks that the number of arguments is correct and that the arguments are of correct type. If a runtime error would occur, `psd-apply` calls the debugger command loop. Runtime errors that occur in non-debugged code can not be caught this way.

There are still some cases in the current implementation where a runtime error can occur. For example, for the `assoc` procedure, the second argument should be a list of lists. Currently, it is only checked that it is a list.

8 Tail Recursion and Continuations

In Scheme, iteration is expressed as tail recursion. It is important that the debugger maintains this property whenever possible, because otherwise a debugged program might easily run out of memory. During single stepping Psd does not preserve tail recursiveness (because of the way single stepping is implemented), but in other situations it is preserved.

Tail recursiveness could be fully preserved by using breakpoints to implement single stepping. This would add some complexity, though, and since it would take quite a time to run out of memory by single stepping a program by hand, it has not been judged worth the effort.

First class continuations are not a problem, since they are handled by the underlying Scheme environment.

9 Caveats and Limitations

Psd shares the problem common to all debuggers: running the debugged program is not exactly the same as running the same program without the debugger. Psd tries to be true to the underlying environment, but there is at least one aspect that would require access to the underlying implementation: evaluation order.

In order to catch runtime errors, Psd transforms each procedure call into a call to the procedure `psd-apply`, with the subexpressions of the original combination as arguments. Because of the way the Scheme language is defined, it cannot be guaranteed that the evaluation order of the subexpressions is the same in the debugged program as in the original program. There is not much that can be guaranteed about the order of evaluation anyway, which makes this a non-issue for well written programs. In practice, however, debuggers are used for finding bugs in ill behaved programs, so it should be addressed somehow. If an implementation always uses a left to right or right to left evaluation, Psd preserves the evaluation order.

Another limitation is caused by the lack of a standard method for accessing top level variables. Psd provides access to all variables defined in the files that are being debugged, but all other top level variables are inaccessible. A partial solution would be to detect all nonlocal variables that are referenced in the debugged expressions. This may be implemented in future versions of Psd.

Conventional debuggers can provide some help even when an error happens in a part of a program that has not been compiled for debugging. With Psd this is not possible, because debugging with Psd relies on the correct execution of programs.

A yet unsolved problem is providing the user with backtrace information. Access to local variables in the current lexical context is easy to provide using closures, but access to nonlocal variales at the calling procedure is more difficult. It would be possible to collect backtrace by passing the backtrace as an extra parameter with every procedure call. This is not a very good solution, because it would not allow mixing debugged and undebugged code. Another solution would be to collect the same backtrace by inserting assignments to a global variable at each procedure entry and exit. This approach breaks when `call-with-current-continuation` is used, because a procedure invocation can be exited an arbitrary number of times.

An inherent problem with instrumenting the original source code is that the resulting instrumented files are quite large. The extreme case is the one line procedure

```
(define (foo x) (+ x 1))
```

that is expanded from 25 bytes to 963 bytes, giving an expansion factor of 39. A more typical case is the instrumentation code of Psd that was expanded from 18058 bytes to 259309 bytes, giving a factor of 14. The time taken to instrument the instrumentation code was little over a minute on a Sun Sparcstation SLC using Aubrey Jaffer's scm interpreter.

The instrumented code is so much slower than the original that it is by no means practical to instrument all the procedures of a large application. A binary tree implementation was instrumented, and the slowdown caused by instrumentation was in the range of 170–290. Usually the problem can be pinpointed to a few procedures with a fair accuracy without a debugger, though, and the debugger can be applied only to them. In practice the slowness has not been a serious problem.

10 Availability of Psd

Psd is available from the author using email, or from `cs.tut.fi` as the file `/pub/src/languages/schemes/psd-1.1.tar.Z` using anonymous ftp. Psd is placed under the GNU General Public License,* so it can be freely used and distributed.

References

[1] Jurgen Heymann. A 100 % portable inline-debugger. *Sigplan Notices*, 28(9):39–46, September 1993.

[2] Daniel LaLiberte. The edebug package for emacs lisp in the GNU Emacs distribution.

[3] Jonathan A. Rees and William Clinger, editors. The revised[4] report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July–September 1992.

* The General Public License is included in the Psd distribution, or it can be obtained from the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA