Psd – a Portable Scheme Debugger

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

July 9, 1992

1 Introduction

This documents describes the Portabe Scheme Debugger, henceforth referred to as psd. Psd is a Scheme debugger based on instrumenting source code. It uses only features described in R4RS [1], so it should run on any R4RS compliant implementation. It has also succesfully been run on a R3RS implementation as well.

The motivation behind writing psd was the poor support for debugging found in many Scheme implementations. Lisp is often boasted to have the best programming environments around, which can well be true, if you can afford a commercial Common Lisp. Once you get to the free implementations, though, there is very little compared to what debuggers like gdb can give you in the C world. (To cover my back, I'll have to admit that I have not had time to look into all possible free Scheme and Common Lisp implementations, for example the CMU Common Lisp.)

One of the main abilities that Scheme debuggers usually lack is source level debugging. For a seasoned lisp hacker a typical lisp debugger with a read-eval-print loop and access to the call stack can be very effective. For novice programmers, however, who do not have a very good idea of what they are doing in the first place, it can be very confusing. I have been teaching an introductory programming course and a data structures and algorithms course in Tampere University of Technology for some years, and I have tried to keep my students in mind when writing psd.

2 Design Decisions

I have used several Scheme implementations in the past, and I expect to be using several more in the future. Therefore, I did not want to tie psd into any

particular implementation. This causes several drawbacks. For example, it is not possible to look into the call stack to provide backtrace information, variable access must be done in a very roundabout way etc. The approach psd takes is to debug a procedure by instrumenting the original source code.[1]

I wanted psd to be just an additional tool for the programming environment, not an environment in itself. This means, for example, that a procedure must maintain its interface even though it is being debugged. This way, one can use psd only on the offending procedures, and run the others using the underlying implementation. If the implementation provides a compiler, the debugged procedures can be compiled.

Experience with the way gdb interacts with GNU Emacs shows that this kind of cooperation can provide a very flexible environment. Source level debugging with psd is therefore done the same way: the debugged program emits specially formatted position information, which Emacs tracks and maintains an arrow that points to the current source line.

3 Basic Ideas

Before going into implementation details, I'll try to give an overview of the components that make up psd, as well as what happens when a procedure is debugged.

Psd consists of three components: some Scheme code for instrumenting the source, runtime support written in Scheme (the debugger command loop) and some Emacs Lisp code for interfacing with Emacs. The instrumentation code is loaded in the same Scheme session as the debugged program, but it could as well be run as a separate process. The runtime support code must be loaded in the same session as the debugged program.

Psd adds some functionality to an existing package (`cmuscheme.el` by Olin Shivers) that supports running an inferior Scheme session under GNU Emacs. When the inferior Scheme buffer is put into psd mode, the instrumentation and runtime code are read into the Scheme interpreter. The user can now request for individual procedures or whole files to be debugged.

The instrumentation code always works on a per file basis. When a single definition is picked from a source file, it is written into a temporary file with information about where it originally came from. The instrumentation code reads a Scheme source file and produces an instrumented version of it. The instrumented code is then loaded into the Scheme interpreter. The details of issuing the commands, generating temporary file names etc. are handled by the Emacs Lisp code.

[1] There is at least one other instrumenting debugger, namely the edebug package for debugging GNU Emacs Lisp, written by Daniel LaLiberte. Undoubtedly there are more that I don't know of.

From the outside, the debugged procedures are equivalent to their undebugged versions, so they can be invoked either from the top level or from within other procedures. The user can set breakpoints by issuing commands in Emacs buffers containing Scheme code.

When the instrumented code is run, it calls the psd command loop every time it is about to evaluate an expression. The command loop is called with position information, procedures for accessing variables and the original expression packaged up in a lambda form. The command loop takes control, and lets the user examine variables etc. When the user wants to continue, the command loop calls the procedure containing the wrapped up expression. Because this expression is also instrumented, it calls the psd command loop again and so forth. Each invocation of the command loop returns the value that the wrapped up expression returned.

When the command loop is called, it checks whether it should just call the wrapped up expression or break. It stops the execution of the procedure and prompts for user commands, if it is in a stepping mode, or if the user has set a breakpoint on the current line.

4 Run Time Errors

A very convinient way to use a debugger is to let the program run until a run time error occurs, and then examine the state of the program. Psd is different from usual debuggers in that it relies on the normal execution of the debugged program. If there is a run time error, the user is either put into the native debugger of the implementation or back to the top level. Run time errors must therefore be detected before they happen.

There are two kinds of errors: calling a procdure with a wrong number of arguments and calling it with arguments of wrong type. Although these can both be viewed as type errors, there is an importand difference. Calling any procedure with the wrong number of arguments causes a run time error, whereas calling a procedure with a wrong type of a arguments causes problems only if the procedure is a primitive procedure. Primitive procedures are known in advance, so they are fairly easy to handle. Checking the number of arguments for user procedures would require monitoring all text that is sent to the Scheme process, and it is not currently done.

Psd detects run time errors by replacing each procedure call with a call to the procedure `psd-apply`, with the values of the subexpressions of the original procedure call. Before applying the procedure to its arguments, `psd-apply` checks that the procedure indeed is a procedure. If it is a primitive procedure, the number and type of arguments are also chekced. If a run time error would occur, the debugger command loop is called.

5 Inherent Limitations of the Instrumenting Approach

There are some problems with the instrumentation approach to debugging. Maybe the most important one is the difficulty of accessing dynamic information, i.e. bactrace information.

It would be possible to give the user access to the backtrace information, by passing an extra parameter with each procedure call. The debugger would have access to all the local variables in the call stack as well as information about where the execution came from by using it. This would, however, require that all procedures would accept an extra argument, which conflicts with the tool idea of psd. One could then not simply load a Scheme file and use it, or write a definition directly to the Scheme top level. This approach would also practically destroy the benefit of tail recursion, because every call would use some memory even in a tail call position.

Another approach to saving backtrace information would be to add the information in front of a globally visible list every time a procedure is entered, and removing it when the procedure is exited. This way, the debugged procedures would retain their interface, but tail recursiveness would be lost. This approach also breaks if one uses `call/cc`, because it is possible to enter or exit a procedure without passing thru its whole body. Without `call/cc` this approach would work, though, and it is possible that psd will have it in later versions.

Because the instrumented programs and the runtime support for the debugger live in the same name space, there are some names that can not be used in the debugged programs. In psd, all the globally visible procedures start with the prefix `psd-`, and variables with the prefix `*psd-`.

6 Limitations of the Current Implementation

The current version handles all syntactic forms except `=>`, `delay` and unquoting. Unquoting is supported in the sense that procedures containing quasiquote and unquotations can de debugged, but it is not possible to step thru an unquotation, or set a breakpoint within a quasiquotation.

The reader understands symbols, boolean values, strings, vector, characters, integers, simple floats and lists. Fancier numbers like complex numbers etc. are not supported. They are not very hard to implement, they are just not on top of the priority list for me. Hex, octal and binary numbers do work, though, thanks to Edward Briggs.

The instrumented files are quite large, which may also be a problem. Typically, however, one is interested in only a small subset of the program at a time, so in practice this is maybe not significant.

One thing that psd is not guaranteed to preserve is the order of evaluation. Because of the additional code that psd adds to the program, it is possible that the instrumented version of a procedure call is evaluated in a different order than the original. If the Scheme implementation used evaluates all arguments from left to right or right to left, there is no problem. If, however, the order of evaluation is something more exotic, the order of evaluation may change. In practice this is probably not a problem, and you deserve all the problems you get if you write code that depends on the order of evaluation.

7 Instrumentation

Debugging with psd is accomplished by instrumenting the original source code with calls to the debugger. The current environment is passed to the debugger in such a manner that it can be examined and modified. The debugger is called before evaluating each expression.

7.1 Manipulating the Environment

One problem with a portable debugger is that there is no standard way to access and mutate the variable bindings from "outside" using the names visible in a given scope. In psd this has been solved by inserting two procedures in all places where variable binding takes place. These procedures, `psd-val` and `psd-set!` perform the mapping between symbols and the variables they represent. For example, a `let` form binding the variables `x` and `y` would be instrumented as in figure 1.

Note how the scope rules come for free: in the environment where `psd-val` and `psd-set!` are defined, the correspondign procedures from the surrounding scope are visible. The instrumented program also contains global definitions of `psd-val` and `psd-set!` that allow the user access global variables defined in the program.

7.2 Instrumentation of Scheme Forms

The syntactic forms that need special consideration are: `and`, `begin`, `case`, `cond`, `define`, `do`, `if`, `lambda`, `let*`, `letrec`, `let`, `or`, `quasiquote`, `quote`, `set!` and the procedure call.

The basic strategy is to wrap each expression within a `lambda` form, effectively delaying its execution. This closure is passed to the debugging procedure, which allows the user to examine and manipulate the environment before continuing with the program. Continuing is accomplished simply by calling the closure.

The debugger also gets the name of the source file and the starting and ending positions of the current expression. This is used for showing where

```
(let ((x 1)
      (y 2))
  <code that uses x and y>)

(let ((x 1)
      (y 2))
  (let ((psd-val
         (lambda (sym)
           (case sym
             ((x) x)
             ((y) y)
             (else (psd-val sym)))))
        (psd-set!
         (lambda (sym val)
           (case ssym
             ((x) (set! x val))
             ((y) (set! y val))
             (else (psd-set! sym val))))))
    <code that uses x and y>))
```

Figure 1: Accessing variables by name

the program execution is in the source code. To be able to show the context (the lexically surrounding procedure definitions), a definition for procedure `psd-context` is placed in front of every procedure definition. This might be a bit of overkill, as the context is usually clear from the source code.

For most of the Scheme forms instrumentation is almost trivial. All that has to be done is to instrument each subexpression recursively, and wrap a suitable debugger call around the expression. Forms that introduce new bindings (the `let` variants, `do` and `lambda`) require an additional `let`-wrapper to store `psd-val` and `psd-set!`.

8 Accessing Global Variables

Global variables are handled in a similar manner as local variables. When the psd runtime system is loaded, the variables `psd-global-symbol-setters` and `psd-global-symbol-accessors` are defined. After instrumenting a source file, psd knows the top level definitions for that file. It then writes two expressions to the file. Each one adds a procedure mapping the names of the top level definitions to the actual variables in front of the appropriate list. When the instrumented file is loaded these expressions are evaluated. The top level definitions of `psd-set!` and `psd-val` call the procedures one by one, until one of them indicates success, or all of them fail. In this case they tell the user that psd does not have access to that variable.

9 Breakpoints and Stepping

Once the code is instrumented, breaking the execution at selected places is relatively easy. Evaluation of every expression starts with a call to `psd-debug`, which checks whether the user should be prompted or not. Breakpoints are implemented simply by keeping a list of locations (file name and line number), and checking if the current location is one of them. Stepping is implemented with a flag that tells `psd-debug` whether it should stop before and after every evaluation or not. The Emacs Lisp code takes care of issuing the necessary commands to the inferior Scheme.

A A Scheme Procedure and its Instrumented Version

To give an idea of what the instrumented code looks like, this appendix gives a listing of the result of instrumenting a procedure. The procedure is

```
(define (foo x)
```

```
  (let ((bar (* 10 x)))
    (* x bar)))
```

And its instrumented version is

```
(define foo
  (let ((psd-context (lambda () (cons 'foo (psd-context)))))
    (lambda (x)
      (let ((psd-val (lambda (psd-temp)
                       (case psd-temp
                         ((x) x)
                         (else (psd-val psd-temp)))))
            (psd-set! (lambda (psd-temp psd-temp2)
                       (case psd-temp
                         ((x) (set! x psd-temp2))
                         (else (psd-set! psd-temp psd-temp2))))))
        (psd-debug psd-val psd-set! psd-context
                   '(let ((bar (* 10 x))) (* x bar))
                   2 3 4
                   (lambda ()
                     (let ((bar
                            (psd-debug psd-val psd-set! psd-context
                                       '(* 10 x) 2 3 3
                                       (lambda ()
                                         (psd-apply
                                          ((lambda x x)
                                           (psd-debug psd-val psd-set!
                                                      psd-context '*
                                                      2 3 3
                                                      (lambda () *))
                                           (psd-debug psd-val psd-set!
                                                      psd-context '10
                                                      2 3 3
                                                      (lambda () 10))
                                           (psd-debug psd-val psd-set!
                                                      psd-context 'x
                                                      2 3 3
                                                      (lambda () x)))
                                          psd-val psd-set! psd-context
                                          '(* 10 x)
                                          2 3 3 #f)))))
                           (let ((psd-val (lambda (psd-temp)
                                            (case psd-temp
                                              ((bar) bar)
                                              (else (psd-val psd-temp)))))
                                 (psd-set! (lambda (psd-temp psd-temp2)
                                             (case psd-temp
```

```
                                    ((bar) (set! bar psd-temp2))
                                    (else
                                        (psd-set! psd-temp
                                                  psd-temp2))))))
                          (psd-debug psd-val psd-set!
                                     psd-context '(* x bar)
                                     2 4 4
                                     (lambda ()
                                       (psd-apply
                                        ((lambda x x)
                                         (psd-debug psd-val psd-set!
                                                    psd-context '*
                                                    2 4 4
                                                    (lambda () *))
                                         (psd-debug psd-val psd-set!
                                                    psd-context 'x
                                                    2 4 4
                                                    (lambda () x))
                                         (psd-debug psd-val psd-set!
                                                    psd-context 'bar
                                                    2 4 4
                                                    (lambda () bar)))
                                        psd-val psd-set!
                                        psd-context
                                        '(* x bar)
                                        2 4 4 #f))))))))))))

(set! psd-global-symbol-accessors
      (cons (lambda (psd-temp)
              (case psd-temp
                ((foo) ((lambda x x) foo))
                (else #f)))
            psd-global-symbol-accessors))
(set! psd-global-symbol-setters
      (cons (lambda (psd-temp psd-temp2)
              (case psd-temp
                ((foo) (set! foo psd-temp2))
                (else #f)))
            psd-global-symbol-setters))
```

References

[1] 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.