Massachusetts Institute of Technology

6.945: Large Scale Symbolic Systems

Final Report

Debugging, Better

Kenneth Friedman
Blake Elias
Jared Pochtar

ksf@mit, eliasb@mit.edu, jpochtar@college.harvard.edu

Professor Gerald Jay Sussman
Manushaqe Muco

May 15, 2017

Abstract—To more easily write large scale, robust systems, it is necessary to improve our debugging tools and methods. In this project, we explore new methods of debugging Scheme. By implementing a fully working programming environment (also known as an integrated development environment, or IDE), we demonstrate major improvements to current debugging methods. The IDE contains three major components: a highlight to evaluate system, a way to manipulate values by dragging, and expression investigation.

Here, we first explain the motivation beind our IDE. Then, we describe our environment and how it communicates with Scheme. Next, we describe the major debugging components in detail, and describe their implementation. Finally, we summerize our findings and describe future possibilities.

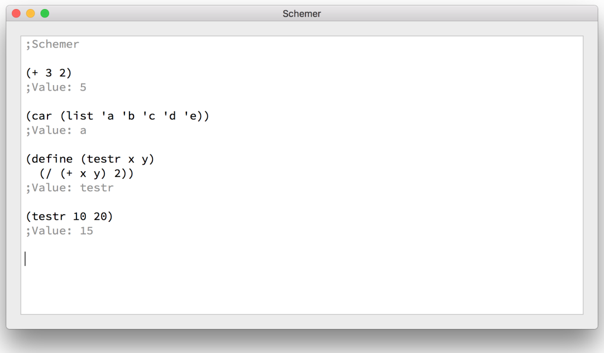
The app is fully open source with an MIT License. The app and the code are available here: github.com/kennethshawfriedman/Schemer.

1 Motivation

"Programs must be written for people to read, and only incidentally for machines to execute" -Abelson and Sussman, SCIP

"Debugging is twice as hard as writing the code in the first place." -Brian W. Kernighan

Kernighan wrote those words in 1978, and they are still true today. Our vision for this project is introduce a brand new Scheme programming environment that makes major improvemnts to debugging. While this is in a prototype state, we have a fully working implementation that can already help you better understand your code.
To take steps towards our vision, we implemented a Scheme IDE with debugging tools built into the environment. We designed our environment with specific design goals in mind. We also have three major subsystems to our IDE. Here, we will describe our design goals and major subsystems in brief.



```
;Schemer

(+ 3 2)
;Value: 5

(car (list 'a 'b 'c 'd 'e))
;Value: a

(define (testr x y)
  (/ (+ x y) 2))
;Value: testr

(testr 10 20)
;Value: 15
```

Fig. 1. A Screenshot of *Schemer*, the working name for our debugging IDE. The graphical user interface handles the input and output of Scheme execution. Similar to using *Emacs* with MIT-Scheme, it is not a REPL. The user is free to place their cursor at any point in the document and execute commands anywhere and in any order. Like building with mud, the user has the freedom!

1.1 Design Goals

Our main goal is to demonstrate that new ideas and techniques in debugging can improve the overall programming experience. In terms of design goals, we followed the major guidelines of the project. Our "mix and match pieces" are the distinct debugging tools, which can be swapped and replaced without harm to the system. Our system is also robust, and incremental changes to the tools will only require incremental changes to the code. Each major system subsystem can be swapped for a different implementation. Finally, the project is well documented, and can be easily run and modified, without the need for dependencies.

1.2 Subsystems

Each subsystem in our IDE (which has the working name: *Schemer*) is a debugging feature. We have three main debugging features.

First, our system includes "Highlight to Preview". This introduces a new concept of a preview-execution. You can temporarily see the output of your code (by selecting a section of text), and then instantly "undo" the computation (by deselecting that text).

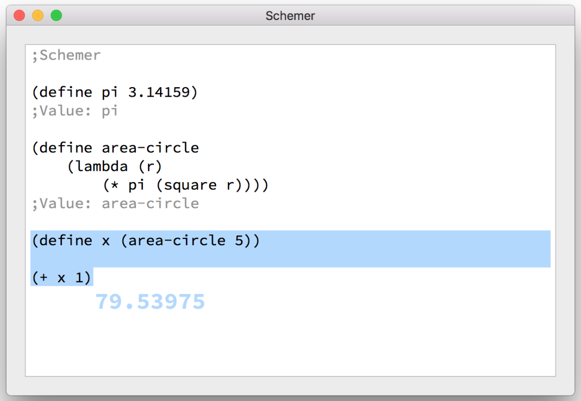
Second, our system includes draggable manipulation. This alows you to easily see the output of a procedure over a range of numerical inputs. Simply hover over a numberical value, and "drag" by pressing and panning the pointer to the left (to decrease the value) or right (to increase the value). The function will automatically compute based on the new value.

Third, Schemer contains a method to explore expressions, known as Expression Investigation. This allows a user to slowly drill down into a procedure, to understand it's "trace," without having to use breakpoints or mannual investigations.

2 Schemer, the IDE

The base level IDE (which is everything before the debugging tools) includes two major components, a "frontend" Mac app GUI, and a "backend" pipe to an MIT Scheme REPL process.

The Mac app GUI gives the appearance of a straightforward text field. It takes input from the keyboard and watches for interactions (such as highlighting, cursor movement, and execution commands). It displays the input from the user and also sends the input to the the backend pipe. Finally, the frontend receives a stream of data from the backend pipe (which is the result of Scheme computation), and then displays the result to the user. The GUI also provides syntax highlighting: for example, comments are shown in a gray font while code is in black.



```
;Schemer

(define pi 3.14159)
;Value: pi

(define area-circle
    (lambda (r)
        (* pi (square r))))
;Value: area-circle

(define x (area-circle 5))

(+ x 1)
      79.53975
```

Fig. 2. The Highlight to Evaluate tool in *Schemer*. Simply select the text you want to execute, and you will see a preview of the value if you were to execute the command. However, the command does not really execute (from the user's perspective). Once they let go, if they were to just run the (+ x 1) command, there would be an error: because x is unbound.

The backend pipe is constantly listening to the frontend, waiting for data. Once data is received, it sends it to the Scheme process that it owns. This backend pipe operates on a seperate thread from the front end, so large computations do not slow the interaction of the frontend. Once the computation is complete, it sends the result back to the frontend to be displayed as text to the user.

Both the Mac app GUI and the backend pipe are written in Swift 3. Swift is an open source language (more information, here: https://swift.org). The Schemer IDE requires no dependencies to run other than the base level OS version: which is MacOS 10.10. It would be possible to port to other operating systems: the native graphic elements of the app are the only thing preventing it from running elsewhere.

3 Major Subsystems / Debugging Tools

Next, we explain the major subsystems to the IDE.

3.1 Highlight to Evaluate

This features introduces a new concept to Scheme programming: a preview execution. A Preview Execution is temporary: It allows a user to see the result of their code, and then immediately undo the fact that the code was executed.

In our debugger, you can simply highlight a section of code, and it will execute a preview. It will show you the result of the execution while the text remains highlighted. However when you deselect the text, any mutated state is automatically removed: as if the execution never took place.

For example, consider the following:

```
(define x 5)
;Value: x

(define x 10)
(+ x 1)

(+ x 3)
```

Fig. 3. Figure 1.3 from Structure and Interpretation of Computer Programs: "a linear recursive process for computing 6!". Our "expression investigation" language is meant to automatically generate output like the one depicted above, in order to help the programmer understand the behavior of their programs.

If the programmer selects the highlighted the 3rd and 4th lines of text above, they will see the value 11. However, if they then highlight the final line, (+ x 3), they will see the value 8, as the environment gets reset to the old value of x, which was 5.

The behind-the-scenes implementation of this involves using Scheme environments, duplicating the environment and modifying its state, and then removing the new environment when the Preview Execution is complete.

3.2 Expression Investigation

This feature allows the programmer to inspect the execution of their program. A programmer may wish to see exactly how an expression evolves into its final value, in a step-by-step manner. In our GUI, the idea (though not yet implemented) is that by highlighting an expression and clicking expand, the programmer can see the expression generated by substituting a functions arguments into the function body. For example with (factorial 4), highlighting the expression will show 24, but clicking the expand button would substitute 4 into the function body of factorial, showing (∗ 4 (fact 3)). From here, the programmer could further expand (fact 3) into (∗ 3 (fact 2)), and so forth.

Before trying to make any of this work in the GUI, we needed to modify the underlying Scheme language to support this. The idea would be to have a language which automatically generates output like figure 1.3 in SICP, reproduced below.

The above-described abilities are not yet complete, but we attempted the following methods:

1) Re-define **eval** and **apply** to print out what is being evaluated or applied at every step. This gives some insight into the program's execution, but for anything larger than the simplest program, it is too verbose and doesn't convey tangible meaning. This is because it displays individual, low-level operations (the "leaves" of the execution tree), without any higher-level information of who made those calls.
2) Re-define **eval** and **apply** to accept a depth argument, which tracks how many more levels deep the recursion should go. When the recursion has reached a pre-determined level, it stops, making **eval** and **apply** just return symbolic expressions (either the expression it was given, or the procedure and arguments it was given). This also involved

modifying **eval** and **apply** to return whatever expressions they were given, in the case when their arguments are only partially-evaluated (and therefore can't actually be computed with). We then had the REPL evaluate the programmer-entered expression multiple times at different depth-limits.

When we tried this, we got results like the following:

```
eval> (fact 3)
(fact 3)
(if (< n 2) 1 (* (fact (- n 1)) n))
1
(#[arity-dispatched-procedure 17 *]
     (fact (- n 1)) n)
(#[arity-dispatched-procedure 17 *]
     (fact (- n 1)) n)
1
(#[arity-dispatched-procedure 17 *]
     (fact (- n 1)) n)
(#[arity-dispatched-procedure 17 *]
     (fact (- n 1)) n)
3
6
```

This had the flaw that argument names would not have their values substituted in (though we think that this would be possible to fix). But perhaps more importantly, this approach would only show a single level of the call chain, substituting the body of a procedure where it was being applied (as in (∗ (fact (− n 1)) n)), but we would never get back an expression showing multiple levels of execution at once, like (∗ (fact 2) 3).

3) Next, we tried having every **apply** call return not just a value, but the value as well as the expression which generated that value (the procedure and arguments). Subsequent **apply** calls would then have to pull out the car of all the arguments in order to get just the values to operate on, ignoring the "history" that was being saved. The idea here is that you could see a tree of your program's execution, with the final value as the root, from where you could then expand different branches to see where different pieces of the final value came from.

At a REPL, the programmer could type the following and get the following result:

```
eval> (+ (1 ()) (2 ()))
(3 (#[arity-dispatched-procedure 15 +]
     (1 ()) (2 ())))
```

The above syntax says, apply "+" to the arguments "1" and "2" that both have no history. This returns an answer of 3 with history showing what function call generated that value.

For more complicated programs, this would show a more interesting output. It might be difficult to parse beyond a couple levels, so we would probably truncate the history beyond a certain depth, leaving it to the programmer to decide which ones to investigate further.

Unfortunately, we were only able to get this behavior to work for the application of strict primitive procedures, but not for compound procedures (this should be quite feasible though, with a bit more debugging which is still planned, by the group members who are still working on this project).

While we mainly tested the above approaches using simple functions like factorial, we see this language feature being especially useful for debugging large systems. In fact, this approach is overkill simple programs – it is easier to just read the code, or evaluate some small examples in the REPL. The real power of this language feature is in debugging large systems. When you have long, complicated code, it is difficult to keep track of what intermediate calculations are being done, and how one function interacts with the rest of your code. So, rather than having to trace it all out in your head, or get your program to crash and then inspect the call trace with the debugger, you would be able to just run the program and then navigate through exactly how it got its answer.

3.3 Draggable Manipulation

Any numerical literal value in Schemer becomes draggable when the programmer hovers on that value in the source code, indicated by a double-arrowed cursor. If the programmer subsequently clicks and drags left or right, the literal value changes and the output (displayed on the right-hand-side) changes along with it. Dragging left makes the value decrease, and dragging right makes it increase, preserving the original type (integer or floating point are the types supported currently).

There is support for future extensions of this functionality to other types as well. For example, strings or lists could also be made draggable. The programmer would just have to define what it means to increment and decrement those types: add or remove one character from the string, add repeated elements to the list, etc.

4 Contributions

Here, we have demostrated a prototype, yet working, IDE for Scheme that includes debugging tools to enhance the programming experience. We introduced three major subcomponents and explained their features. In the future, debugging will not be as hard as it was for Kernighan at Bell Labs.

5 Downloadable Version

Our project is open source and available for download here: github.com/kennethshawfriedman/Schemer.

6 Acknowledgments

The authors would like to thank Professor Sussman and Manushaqe Muco for the adventures.