SchLint

Debugging tools for MIT Scheme

Dylan Sherry dsherry@mit.edu
Joe Henke jdhenke@mit.edu
Owen Derby oderby@mit.edu
March 31, 2013

Introduction

With this project, we set out to expand the debugging facilities available for MIT Scheme. In particular, we wanted to detect as many bugs as possible before runtime, track executed code during runtime for later review, and determine where returned values originated from and what processes modified them. Such advanced debugging utilities provide powerful tools to the programmer, enabling faster iteration by catching bugs sooner and making it easier to find bugs/problems.

To this end, we propose **SchLint**, a simple tool for advanced debugging of MIT Scheme programs. The name is a combination of Scheme and "lint," reflecting one of it's primary purposes as a static analysis tool. With SchLint, we provide tools for all three debugging capabilities outlined above. Our purpose is not to replace any of the existing debugging features of Scheme. In particular, if there are bugs which are caught already easily detected (e.g. syntax errors), we do not address them. Rather, SchLint can be viewed as an augmentation to the standard debugging process in Scheme. It will warn you about possible problems with your code, and aid you in triaging errors you may encounter.

We continue by giving an overview of how SchLint is constructed. Then we proceed into the details of the two major sub-systems of SchLint: 1) the lint-like static analysis and program flow abstraction and 2) the provenance tracking system for scheme objects.

System Overview

SchLint is implemented as a custom MIT Scheme interpreter running within standard MIT Scheme. Instead of starting from scratch, we built SchLint from an existing interpreter[1]. This interpreter provides a REPL environment where the evaluation step occurs in two distinct phases. First, the raw s-expression is analyzed and compiled into a combinator. A combinator is a function which takes one argument, an environment. A combinator may call other combinators. Second, the combinator is called with the current environment to complete the evaluation, and returns a value.

Within this guest interpreter framework, the static analysis is implemented by instrumenting the analyze procedure, to statically analyze the code as it's compiled. To properly detect bugs in the code, SchLint builds a special kind of graph, called a Control Flow Graph (CFG), to model the program code. Once this graph is constructed, bugs can be detected easily by searching for

[1] http://groups.csail.mit.edu/mac/users/gjs/6.945/psets/ps04/code/

certain properties of the graph. After construction of the CFG, the code can be executed[1]. During execution, the CFG is traversed, and the execution is tracked as a trace over a series of edges of the graph. This trace can be queried later, to check for errors or to see code coverage.

Provenance tracking was implemented by slightly changing what sort of object the evaluation passes around internally. Normally, the interpreter passes around values (or lists, or other primitive or user-defined datatypes) while evaluating an expression, and returns that after the evaluation ends. To allow for tracking of arbitrary provenance, we abstracted objects into **cells**. A cell represents a data object and a list of tags. Tags are used to record provenance. With the introduction of cells, everything passes around cells behind the scenes of the interpreter, and the REPL simply extracts the value from the result of an evaluation before returning to the user.

Lint and Scheme

The term “lint” originates from a program out of Bell Labs for statically checking for suspect code in C programs[2]. Nowadays, programs exist which implement lint-like behaviour for most almost every programming language, although none exist for MIT Scheme. With static analysis of supplied code, our linter detects

1. Unused procedures: procedures which are defined but never evaluated.
2. Unreachable procedures: procedures which are defined but can never be evaluated because of where or how they are defined.
3. Undefined procedures: procedures for which there exist one or more calls, but are not defined within the environment of those calls.

To perform this static analysis, we represent the program as a Control Flow Graph (CFG). A CFG attempts to capture all the possible valid execution traces through a program. By capturing what’s allowed, it becomes easy to check if a proposed trace is not allowed. A CFG captures procedure definitions and call sites, which is sufficient to model flow of execution through a program. Using this CFG, we are able to statically detect common bugs in a program and then track how the program is executed at runtime.

Graph Abstraction

To start, we created a simple directed graph library to support building a CFG while abstracting out the details of what constitutes a graph. A graph is simply a collection of node objects and edge objects. Each node object corresponds to a node in the graph, and can have an arbitrary data object associated with it. Each edge object represents a directed edge in the graph, originating at a single node and ending at a single node. Each edge object has a reference to a source node object and destination node object and can have an arbitrary data object associated with it. Node objects maintain a list of all incoming edge objects (edge objects which have that node as a destination) and a list of all outgoing edge objects (edge objects which have that node as a source).

This graph library provides a minimalistic API for creating and manipulating these objects.

---

[1] Because this is implemented as an interpreter, the analysis and execution is actually interleaved. As illustrated by the examples later on, the conceived use case is defining all necessary procedures and structures first, then checking for bugs, then executing.

[2] https://en.wikipedia.org/wiki/Lint_%28software%29

1. A new graph object can be created, and edge and node objects can be created and added to it.
2. The list of all node and edge objects (separately) can be retrieved from the graph, and the existence of a node or edge object can be checked.
3. Incoming and outgoing edge objects can be retrieved given a node object. Conversely, given an edge object, the source and destination node objects can be retrieved.
4. The data associated with edge and node objects can be retrieved and modified.
5. Edge and node objects can be removed from the graph.

Control Flow Graph

We use our graph library to represent the CFG. Nodes in the CFG correspond to procedures in the program under analysis. A CFG encapsulates two unique relations, which are represented as edges:

1. Definitions: "*A* defines *B*" iff the definition of procedure *B* is within the definition of procedure *A*.
2. Applications: "*A* calls *B*" iff there is an application of procedure *B* within the definition of procedure *A*.

Every procedure has a single incoming definition edge. Top-level procedures aren't defined by another procedure, so a fake node, called **root**, is added to be the "procedure" which defines all top-level procedures. A call edge ending at procedure *A* exists for every application of *A*. This means there may be multiple call edges from one node to another (when *B* invokes *A* multiple times) and (multiple) call edges from a node to itself (*A* recursively calls itself).

To construct the CFG, the code is analyzed in a depth-first manner. As the interpreter analyzes the code, it keeps track of the scope in which new definitions will occur by passing around a reference to the current "parent" node, which is initially the root node. As it encounters "define" statements which define a procedure, a new node is added to the graph and a definition edge is added from the current parent node to the new node. The definition expression is then analyzed, with the new node as the parent.

As applications of procedures are encountered, a new call edge should be added from the current parent node to the invoked procedure. Before the edge can be added, however, the node corresponding to the invoked procedure needs to be found in the graph. Given the symbol identifying the procedure and the parent node, the task is to find the corresponding node, if any exists, by searching upwards from the parent. We only need consider the subset of nodes in the graph which are callable from an object in the parent's scope (which is where the application is occurring). Figure 1 gives an illustration of this distinction. We have 3 possible results from our search

1. The node exists and is callable from the current scope.
2. The invoked procedure is defined in the default REPL environment (primitives, SchLint reserved procedures and other procedures defined before Schlint REPL is started)
3. The node does does not exist or is not callable from the current scope.

In the first and second cases, a call edge is added from the parent node to the found node. In the third case, a special "undefined" node is created (or used if already existing), which indicates that such a procedure has been invoked, but not defined, and a call edge is added to it. This allows us to properly track calls to procedures which may be defined later on. When the

procedure is defined in the future, the defined node simply assumes all call edges to this temporary one.

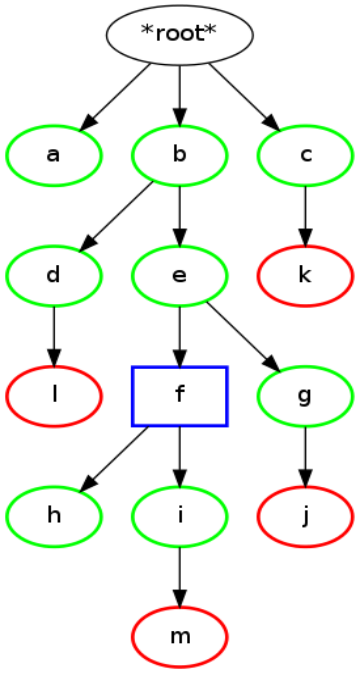


Figure 1: Illustration of callable (green) and un-callable (red) nodes in the CFG from the scope of f.

Detecting Problems

We can use the CFG to statically detect three common bugs in code:

1. Unused Definitions: A procedure which is defined in a non-global scope but which is never called. We can detect this from the CFG because it will be a node or sub-tree which has a define edge, but no call edges to it.
2. Undefined Reference: An invocation of a procedure which doesn't exist. We can detect this by searching for any "undefined"-type nodes.
3. Dead Code: Procedures which are defined but all references to them are lost, such as defining a new procedure with the same name. We can detect dead code by searching the graph for sub-graphs which we can not reach from the root, by following either call or define edges.

Tracking Executions

Aside from statically catching bugs, we use the CFG to track program executions during runtime. We treat executions as a property of call edges, and for each call edge we add an execution event/entry every time that call is actually made at runtime. Each execution event records the arguments for that call and the value it returned. The effect of this is that for any call edge in the CFG, at any point in the runtime, we can stop and see exactly how many times it was actually invoked, with which arguments, and what was returned. Further, we can see which call edges were actually used/traversed and which remained untouched. Finally, we can order the execution events by the time they occurred, and use that to form a very detailed stack-trace

for the entire program. This can be very useful to see how and what a program executed, even if it doesn't produce any errors (which is usually the only time you get a stack trace).

Examples

Here we provide some brief demonstrations of our static analysis system detecting some common problems. For each listing, we run the static analysis by loading in SchLint (via the command (load "load")) and initializing it (via the command (init)), then importing the listing, and finally calling the analysis procedure (check-cfg). For each example, we provide the listing code, the output from the analysis, and the CFG generated (for reference).

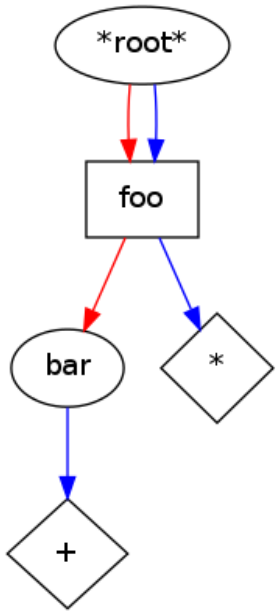
input:

```
(define (foo x)
  (define (bar y)
    (+ y 1))
  (* x x))
(foo 3)
```

output:

```
1 ]=> (check-cfg)
"function bar is unused"
;Value: #f
```

generated CFG:



Listing 1: Detecting unused code

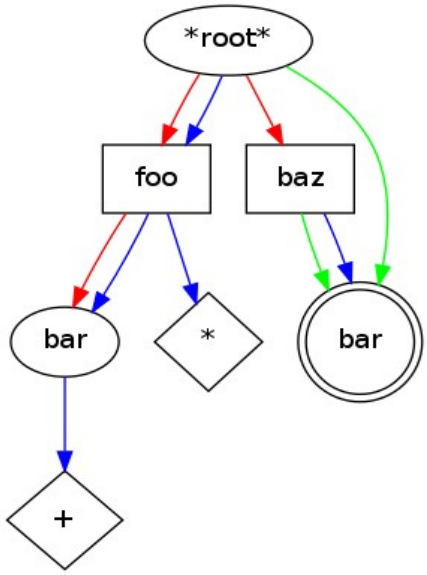
input:

```
(define (foo x)
  (define (bar y)
    (+ y 1))
  (bar (* x x)))
(define (baz x)
  (bar x))
(foo 3)
```

output:

```
1 ]=> (check-cfg)
"baz calls undefined function bar"
;Value: #f
```

generated CFG:



Listing 2: Detecting undefined references.

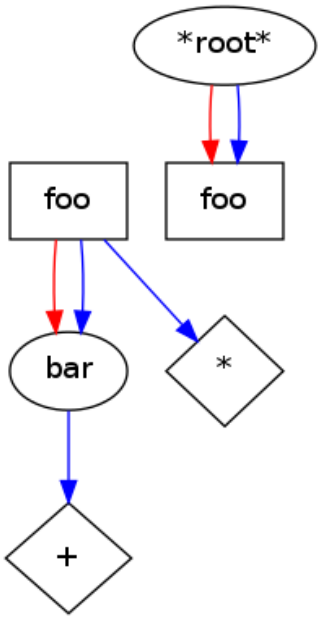
input:

```
(define (foo x)
  (define (bar y)
    (+ y 1))
  (bar (* x x)))
(define (foo x) 2)
(foo 3)
```

output:

```
1 ]=> (check-cfg)
"function foo is unreachable"
"function bar is unreachable"
;Value: #f
```

generated CFG:



Listing 3: Detecting dead code.

Provenance Tracking

The Idea

Provenance tracking refers to seeing what pieces of data contributed to the formation of other pieces of data. From a debugging standpoint, it can immediately answer I have a problem with my output. Which of these inputs need I concern myself with?

More specifically, any object can be given any number of tags. To propagate these tags, objects created from the output of primitive procedures are assigned the union of the tags of the arguments to the primitive procedure. The only other way to propagate tags is through tagging functions themselves. If a function has tags, the object returned from each function call also has that function's tags.

The Interface

We've provided the ability in our guest interpreter to tag objects and see to which objects they then contribute.

Creating Tags

Tags can be attached to any object in any scope.

```
(define a 2)

;;; To user defined variables
(add-tag a 'apples)
```

```
;;; Within functions
(define (foo x)
  (define b 3)
  (add-tag b 'bananas)
  (+ x b))

;;; To primitive functions
(add-tag + 'bar)

;;; Even to constant objects
;;; Note: These are transient objects
(add-tag 1 'baz)
; Value: 1
```

Checking Tags

Now we can check to see how tags have been modified for a particular variable or object after some computation.

```
;;; Check variables
(get-tags a)
; Value: (apples)

(define c a)

;;; Definitions propagate tags
(get-tags c)
; Value: (apples)

;;; Function calls propagate tags
(get-tags (foo a))
; Value: (apples inner bar)

;;; Assigning to transient objects work
;;; but are exactly that - transient
(get-tags (add-tag 1 'one-time))
; Value: (one-time)

(get-tags 1)
; Value: ()
```

Implementation

In summary, we built off of PS4, abstracted the notion of an object to also hold provenance information and ensured the provenance information propagated through newly created objects.

PS4 Recap

We built off the guest interpreter provided in PSET 4 - Compiling to Combinators.

This provided a REPL environment where the evaluation step occurs in two distinct phases. First, the raw expression is compiled into a combinator using analyze. A combinator is a function which takes one argument, an environment. A combinator may call other combinators. Second, the combinator is called with the provided environment, returning a value in scheme. Thus, we have defined our evaluation procedure.

Modifying eval's meaning

The fundamental change we've implemented is to the eval procedure paradigm. It is still coded to be the same thing, however it is now defined semantically to return a cell. A cell represents an abstraction of any kind of scheme object. We've defined a cell to be the following structure.

```
(define-structure cell value tags)
```

So a cell holds two datatypes, but could easily be extended to hold more.
The first component is value. This is an MIT Scheme value and these are the primitives upon which the user of this guest interpreter can build.

The second component is tags. A tag can be any object. In practice they are typically symbols as they are distinguished by eq?. Tags are either added explicitly to an object via add-tag or when the object is created through a primitive procedure, in which case it's tags are set to the union of the arguments to the primitive procedure.

This required removing the assumption that the output value of eval was an MIT Scheme object from the code. This mostly affected all the analyze procedures and the mechanism for variable and environment management. The print step in the REPL environment was also modified to only print the value of the cell that was ultimately returned from eval.

Tail Recursion

Special care was taken to avoid breaking tail recursion within the guest interpreter. If within definitions of analyze, a combinator was called and then tags were added to that result, it would break tail recursion. To avoid this, it is determined if the current function call is in a tail recursive position. If so, it first adds the tags which are to be applied to a queue of pending tags, then makes an appropriate tail recursive tail without adding tags. When a non-recursive call is found, it then adds the queued tags to the result. Top level calls are guaranteed to not be in tail recursive positions and so will inherit the pending tags upon the final return.

Design Decisions

Functions

What tags should be propagated to newly defined compound procedures? We decided none. The only real alternative would be to use the union of all variable accesses made within the function to outside the environment frame created in the function, but those accesses don't occur when the function is defined. Furthermore, any output from the function which was affected by these external variable access will still reflect these sources in their tags.

Because primitive procedure definitions were not interpreted by the guest procedure, it is treated as a black box and it is assumed all arguments contribute to the output. In fact, this may not necessarily be true though, but we could not find a general solution to this.

What if a user explicitly tags a variable which is itself a function? We decided that these tags should in fact be propagated with the output of any successive function calls, to answer the question which objects were affected by this function?

Conditionals

Should the tags of the predicate of a conditional be propagated to the data created in the consequence or alternative? We decided no. This seemed too indirect a connection to include with provenance information, as we intended to represent who contributed to the values of a variables creation, not necessarily just the fact that it was created.

Doubly Linked Tags

Something which was not implemented but we feel would be a great addition to this system is to be able to answer, to whom did I contribute? Currently, any object's tags can be retrieved, but once an object is tagged, one can't easily see what objects now have those tags. It is unclear what such functionality would return. Perhaps any variables bound to objects who has those tags and are defined in the current scope, but it should be investigated further.

Notes on Abstractions

Abstracting Values

We found that even though all references to values are now replaced with cells, the only places from the original ps4 code where a cell must be unpacked down to its value was when applying a primitive procedure, conditional predicates or generic dispatch predicates for analyze. Given this very limited rep-exposure of a value/cell, we feel that choosing this abstraction was a good choice and perhaps even further clarifies the distinction between the interpreter objects, MIT Scheme Objects used to run the interpreter itself and the primitive MIT scheme objects which the interpreter may access.

Abstracting Environments

In the original ps4 code, useful abstractions were defined to abstract away the internals of how an environment was represented, but were not actually used. We replaced all instances of environment rep exposure by using the function calls specifically design to operate on environments. This way, if the structure or internals of an environment must ever be changed, only these operation need be changed.

Conclusion

Debugging in Scheme can be a difficult process, especially if the problematic bug is subtle or involves multiple components. We set out to improve this experience with MIT Scheme. Starting from the ps4 code as a baseline, we constructed SchLint, a collection of useful debugging tools for MIT Scheme.

1. By combining a simple graph API with the compiler interpreter, we were able to abstract the flow of a program into a control flow graph. With this control flow graph, we are able to statically detect bugs involving undefined procedures, uncalled procedures, and dead procedures.
2. Further, we use the generated control flow graph to track program execution at runtime, to provide increased visibility and verbosity when examining how a program ran and why it returned the value it did.
3. Finally, to enhance the ability to determine what inputs affected which outputs, we implemented provenance tracking, which allows inputs and functions to be tagged, so you can quickly determine the entire provenance of an output simply by examining its tags.