A Simple Reflective Interpreter

Stanley Jefferson
Daniel P. Friedman

IMSA '92 International Workshop on
Reflection and Meta-Level Architecture
Tokyo, November 4-7, 1992

A Simple Reflective Interpreter

Stanley Jefferson *
Daniel P. Friedman †

Department of Computer Science
Indiana University
Bloomington, IN 47405

Abstract

Procedurally reflective programming languages enable user programs to semantically extend the language itself, by permitting them to run at the level of the language implementation with access to their context. The reflective tower, first introduced by Brian Smith [9, 10], is the principal architecture for such languages. It is informally described as an infinitely ascending tower of meta-circular interpreters, connected by a mechanism that allows programs at one level to run at the next higher level. Various accounts of the reflective tower have been published, including a meta-circular definition, operational definitions, and denotational definitions. We present an operational account of the main aspects of the reflective tower, which we claim is simpler than previous accounts. Our approach is to implement a finite tower where each level literally runs the level directly below it. A complete Scheme implementation is included.

1 Introduction

The principal characteristic of reflective programming languages is that they are extensible. That is, they permit the user to define new language constructs at the semantic level by effectively adding lines of code to the language processor itself. Lisp-based reflective programming languages, such as 3-Lisp [9, 10], Brown [12], and Blond [2], provide such an extension capability by permitting user-level code to be run as if it were interpreter code with access to the interpreter's current expression, environment, and continuation. With this language extension capability, otherwise primitive language constructs, such as catch, throw, call/cc, boundp, and quote, can be defined as user procedures.

The reflective tower, first introduced by Smith [9, 10], provides a unified and coherent architecture for reflective programming languages. The reflective tower is non-effectively described as an infinitely ascending tower of meta-circular interpreters. The interpreter at the bottom of the tower executes user input, and every other interpreter in the tower executes the interpreter immediately below it. The levels are crucially connected by a mechanism that permits a program running at one level to provide code that is to run at the next higher level. Various accounts of the reflective tower architecture have been given including a meta-circular definition [9, 10], operational definitions [4, 1], and denotational definitions based on the meta-continuation concept [12, 2]. We give another account that we believe is simpler and more easily understood than previous accounts.

Our approach is to give a literal model of a finite reflective tower. An interpreter for a small, simplified subset of Scheme [8] is implemented in Scheme. This interpreter is written in its own subset of Scheme so that it can literally run itself. This permits finite towers of the interpreter to be run (albeit inefficiently). Then, a few modifications are made so that the interpreters in a finite tower are connected as in the infinite reflective tower. Once the literal model of a finite reflective tower is understood, the behavior of an infinite reflective tower can be easily understood as the limiting behavior of a finite reflective tower as its height approaches infinity.

The literal definition is similar to the meta-circular definition of 3-Lisp. It differs in that it runs, it is not circular, and it does not reconstruct evaluation and quotation. The literal model is simpler than the operational and denotational models since it models the reflective tower implicitly rather than explicitly. All of the models are interesting and useful because they clarify different aspects of reflection.

The rest of this paper is organized as follows: section 2 gives a brief overview of the reflective tower architecture; section 3 presents a simple interpreter that is capable of interpreting itself; section 4 derives a language extension mechanism for this interpreter; section 5 presents the implementation of our reflective interpreter; section 6 is a comparison with related work; and section 7 is the conclusion.

2 Overview of the Reflective Tower Architecture

The reflective tower is described as an infinitely ascending tower of meta-circular interpreters. The interpreter at the tower's bottom executes user input, and every other interpreter in the tower executes the interpreter immediately below it. Each interpreter state is passed around as three arguments to the interpreter: an expression, environment and continuation. An interpreter running at one level may have a different state than an interpreter running at a different level. Normally, the interpreters share a global environment.

*Supported by grant NSF CCR 89-01919. Current affiliation: Hewlett-Packard Laboratories, 3500 Deer Creek Road, Building 26U, Palo Alto, CA 94304-1392
†Supported by grants NSF CCR 89-01919 and NSF CCR 90-00597.

Reifying procedures provide a mechanism for running code at the next higher level and gaining access to that level's state in the form of an expression, environment, and continuation. A reifying procedure, or simply reifier, has three formal parameters. When a reifier is applied, its body is run as if it were code belonging to the interpreter running the application. The body is run in an environment where the first formal parameter is bound to the list of unevaluated arguments, the second formal parameter is bound to the environment argument of the interpreter running the application, and the third formal parameter is bound to the continuation argument of the interpreter. In a fully metacircular model of the architecture, the body of a reifier has access to the procedures and structures that the interpreter itself has. Since reifying procedures can themselves invoke reifying procedures, it is possible to run user code at arbitrarily high levels of the tower.

3 A Simple Interpreter

We begin by presenting a garden variety interpreter for a subset of Scheme. This interpreter, referred to as $\mathcal{I}$, is written in its own language so that it can interpret itself. It is written in continuation-passing style. The extended BNF for the subset of the Scheme language implemented by $\mathcal{I}$ follows (non-terminals are printed in italic font and terminals are printed in typewriter font).

exp	::=	(*exp* {*exp*}*)
	|	(`lambda` ({*identifier*}*) {*exp*}$^+$)
	|	(`if` *exp exp exp*)
	|	(`set!` *identifier exp*)
	|	(`quote` *exp*)
	|	*identifier*
	|	*constant*

The syntactic categories of identifiers and constants are the same as in Scheme. The remaining notable characteristics of $\mathcal{I}$ follow: `set!` assigns to an unbound identifier; only a small subset of Scheme primitives is implemented; and error detection and reporting is almost non-existent.

Other than expressions, the principal structures manipulated by $\mathcal{I}$ are environments, continuations, and procedures. As usual, a continuation is a procedure of one argument. Environments are represented by association lists. Each element of the association list is a pair consisting of an identifier and a value. The interpreter handles two types of procedures. The two types are represented as lists and are distinguished by a tag. A compound procedure is represented by a list containing a tag, formal parameters, a procedure body, and an environment. The body of a compound procedure is a non-empty list of expressions. The environments contained in compound procedures and passed around by the interpreter only list the bindings that extend the global environment; whenever a variable binding is looked up (via `get-pair`) the global environment is searched if necessary. A primitive procedure contains a tag and an operator symbol. For example, the identity procedure is `'(compound (x) (x) ())`, and the addition procedure is `'(primitive +)`. We follow the convention that `e`, `r` and `k` are expression, environment, and continuation, respectively.

Following are the Scheme definitions of some of the principal procedures that define $\mathcal{I}$. Since the interpreter must interpret itself, the definitions are also $\mathcal{I}$ definitions. All of the supporting definitions are given in the appendix. The procedure `evaluate` dispatches on the syntactic type of its expression operand `e`.

```
(set! evaluate
  (lambda (e r k)
    ((if (constant? e)
       evaluate-constant
       (if (variable? e)
         evaluate-variable
         (if (if? e)
           evaluate-if
           (if (assignment? e)
             evaluate-assignment
             (if (abstraction? e)
               evaluate-abstraction
               evaluate-combination)))))
      e r k)))
```

As usual, an abstraction simply evaluates to a procedure. Following are the details, since a good understanding is required in the next section.

```
(set! evaluate-abstraction
  (lambda (e r k)
    (k (make-compound
         (formals-part e) (body-part e) r))))

(set! make-compound
  (lambda (formals body r)
    (list 'compound formals body r)))
```

Combinations are evaluated in a conventional manner. The operator is evaluated to obtain a procedure, and the operands are evaluated to obtain arguments. Then the body of the procedure is evaluated in the procedure's lexical environment extended by binding the procedure's formal parameters to its arguments.

```
(set! evaluate-combination
  (lambda (e r k)
    (evaluate (operator-part e) r
      (lambda (proc)
        (evaluate-operands (operands-part e) r
          (lambda (args)
            (apply-procedure proc args k)))))))

(set! apply-procedure
  (lambda (proc args k)
    (if (compound? proc)
      (evaluate-sequence
        (procedure-body proc)
        (extend
          (procedure-environment proc)
          (procedure-parameters proc)
          args)
        k)
      (k (apply-primitive
           (procedure-name proc) args)))))
```

`Evaluate-operands` produces a list of arguments from a list of operands. `Evaluate-sequence` successively evaluates a sequence of expressions, yielding the value of the final expression. Apply-primitive simply passes the task of applying a primitive to the language processor running the interpreter.

The global environment of $\mathcal{I}$, bound to global-env, must be initialized with bindings for the primitive procedures.

```
(set! initialize-global-env
  (lambda ()
    (set! global-env
      (extend
        empty-env
        (primitive-identifiers)
        (mapper make-primitive
          (primitive-identifiers))))))
```

A read-eval-print loop is provided by the following procedure.

```
(set! openloop
  (lambda (read-prompt write-prompt)
    (display read-prompt)
    (evaluate (read) empty-env
      (lambda (v)
        (display write-prompt)
        (if (eq? v (void))
            "Nothing will be displayed"
            (write v))
        (newline)
        (openloop read-prompt write-prompt)))))
```

Openloop always prints its output prompt after the input expression has finished running, even though no output is printed when an expression returns a void value.

The following transcript illustrates starting the interpreter and running an expression (">" is the Scheme prompt).

```
> (load "simple.ss")
> (initialize-global-env)
> (openloop "0> " "0: ")
0> (+ 2 3)
0: 5
0>
```

Since the interpreter is written in its own language, it is possible to load the interpreter and have it run itself. The definition of $\mathcal{I}$ includes the following procedure for loading source files.

```
(set! loadfile
  (lambda (file)
    ((lambda(port)
       ((lambda (loop)
          (set! loop
            (lambda (v)
              (if (eof-object? v)
                (close-input-port port)
                (evaluate v empty-env
                  (lambda (ignore)
                    (loop (read port)))))))
          (loop (read port)))
        '*))
     (open-input-file file))))
```

Using, loadfile, we create a two-level tower of $\mathcal{I}$ interpreters:

```
> (load "simple.ss")
> (initialize-global-env)
> (loadfile "simple.ss")
> (openloop "0> " "0: ")
0> (initialize-global-env)
0:
0> (openloop "1> " "1: ")
1> (+ 2 3)
1: 5
1>
```

The absence of a level-0 output prompt after the second openloop indicates that the second openloop has not finished running. We orient the tower in the same way as 3-LISP (i.e., the level above runs the level below), but our level numbering convention is reversed from that of 3-LISP.

$\mathcal{I}$ traps unbound variable errors and exits. Notice how a level is lost in the following continuation of the previous transcript.

```
1> xxx
Error: symbol not bound: xxx
0:
0>
```

The level-1 $\mathcal{I}$ displays the error message and exits. Control then passes to the level-0 openloop, which indicates that level-1 returned a void value by printing a level-0 output prompt with no value following it.

4 Adding an Extension Mechanism

We now consider adding a mechanism to $\mathcal{I}$ that would enable user-supplied code to be run at the level of the interpreter. As it stands, $\mathcal{I}$ never runs user code at the level of the interpreter. Rather, user code is simply inspected, decomposed and manipulated by the code that implements $\mathcal{I}$. The mechanism that we propose to study is a (rather impractical) special form (exit *exp*) that runs its unevaluated operand at the level of the interpreter code and then exits the interpreter. We've chosen exit solely for its simplicity and ability to illustrate some fundamental ideas. We use $\mathcal{I}_{exit}$ to refer to the putative interpreter that extends $\mathcal{I}$ with the exit mechanism. The following illustrates the behavior of exit in a two-level tower.

```
1> (+ 5 (* 3 2))
1: 11
1> (+ 5 (exit (* 3 2)))
0: 6
0>
```

At the very least, a new line must be added to evaluate in order to dispatch exit forms. Assume that in $\mathcal{I}_{exit}$ the following line has been added to the definition of evaluate immediately after the dispatch line for assignment (and that parentheses have been appropriately adjusted).

```
(if (exit? e) evaluate-exit
```

Then, the evaluation of (exit *exp*) should proceed as if evaluate-exit were temporarily defined as

```
(lambda (e r k)
  exp)
```

We need a way of abstracting *exp* from this expression in terms of `e`. This is not directly possible since `lambda` does not evaluate its arguments. Also, we reject as too complicated and inelegant the possibility of dynamically modifying the procedure objects constituting $\mathcal{I}_{exit}$. We proceed by deriving an acceptable definition of `evaluate-exit` from the previous `lambda` expression. The derivation requires the additional assumption that *exp* is closed in the global environment. The preceding `lambda` expression can be written as

```
(lambda (e r k)
  ((lambda () exp)))
```

Since a `lambda` expression evaluates to a procedure (formed by `make-compound`), we may write the above as

```
(lambda (e r k)
  ((make-compound
     '() (list (quote exp)) '())))
```

Finally, we may abstract *exp*, yielding the following definition of `evaluate-exit`

```
(set! evaluate-exit
  (lambda (e r k)
    ((make-compound
       '() (cdr e) '()))))
```

The expression

```
(make-compound '() (cdr e) '())
```

evaluates to a procedure object, say *proc*, of $\mathcal{I}_{exit}$. The problem is that *proc* probably won't be recognized as a procedure object by the language processor running $\mathcal{I}_{exit}$. For example, when only one level of $\mathcal{I}_{exit}$ is run by our Scheme system, the following session ensues:

```
0> (exit (* 3 2))
Error: attempt to apply non-procedure
       (compound () ((* 3 2)) ()).
>
```

However, if the language processor that is running $\mathcal{I}_{exit}$ recognizes the procedure objects of $\mathcal{I}_{exit}$, then this definition of `evaluate-exit` works. Thus, we are led to consider a two-level tower of interpreters, where $\mathcal{I}_{exit}$ is run by either $\mathcal{I}_{exit}$ or $\mathcal{I}$. The more levels there are in the tower, the higher in the tower an `exit` expression can be successfully run. For example, the expression `(exit (exit 17))` fails in a two-level tower:

```
0> (openloop "1> " "1: ")
1> (exit (exit 17))
Error: attempt to apply non-procedure
       (compound () (17) ()).
>
```

On the other hand, `(exit (exit 17))` would run without error in a three-level tower, returning to level 0. Of course, uniformity of the levels would be achieved by modeling an infinite tower as in Brown, 3-LISP, and Blond.

There is nothing mysterious about our final definition of `evaluate-exit`. The expression

```
((make-compound '() (cdr e) '()))
```

in the definition of `evaluate-exit` runs the argument of `exit` at the level of interpreter code because it is an instance of the following *basic observation*:

> If the combination (exp_0 exp_1 ...) is run at level l and exp_0 evaluates to a compound procedure object p, then the body of p is run at level l.

The ability to run user code at the level of interpreter code is a consequence of:

1. the *basic observation*;
2. the capability for user code to create procedure objects dynamically; and
3. the capability of the interpreter to interpret itself, so that a tower could be created.

Item 2 is included because an interpreter running at level $l+1$ is run as user code by the interpreter at level l (using our level numbering for prompts).

In $\mathcal{I}_{exit}$ we have an instance of crossing semantic levels, where *exp* is treated both as data and code. When $\mathcal{I}_{exit}$ interprets the user expression `(exit exp)`, it treats the expression as data: extracting its car, dispatching on its car, passing it as an argument, extracting its cdr, embedding its cdr in a procedure object. But, then the procedure object with *exp* in its body is applied as interpreter code, and *exp* is run as interpreter code by the language processor running $\mathcal{I}_{exit}$. In contrast, $\mathcal{I}$ always treats user expressions as data and never allows user expressions to be run as interpreter code.

The reader may be wondering why we couldn't just define `evaluate-exit` as follows, and avoid the whole issue of towers.

```
(set! evaluate-exit
  (lambda (e r k)
    (evaluate (2nd e) r (lambda (x) x))))
```

This definition does not satisfy our requirement that the argument to `exit` run at the same level as interpreter code. Indeed, with this definition, the argument to `exit` is always treated as data by the interpreter. The following transcript, when compared with the previous transcript, illustrates a behavioral difference between this definition and our derived definition.

```
0> (openloop "1> " "1: ")
1> (exit (exit 17))
0: 17
0>
```

5 A Reflective Interpreter

We now turn our attention to defining a reflective interpreter based on a finite, literal tower. First, we extend the definition of $\mathcal{I}$ in order to solve a couple of technical problems with literal towers. Then, we make some extensions and a minor modification so that the interpreter can handle reifying procedures. We refer to the resulting interpreter as $\mathcal{I}_R$. A complete definition of $\mathcal{I}_R$, including all utility functions, can be found in the appendix.

5.1 A Self-interpreting Interpreter

The method of building towers in the previous sections has two drawbacks. The first drawback is that the global environment is not structurally shared among the different levels of a tower. The global environment at each level of a tower is initialized with the same definitions, but the global environment of each level is entirely constructed using new pairs. Hence, assignments at one level are not observable at other levels. This is illustrated by the following transcript.

```
> (load "simple.ss")
> (initialize-global-env)
> (loadfile "simple.ss")
> (openloop "0> " "0: ")
0> (set! level-0-var 0)
0:
0> level-0-var
0: 0
0> (initialize-global-env)
0:
0> (openloop "1> " "1: ")
1> (set! level-1-var 1)
1:
1> level-1-var
1: 1
1> level-0-var
Error: symbol not bound: level-0-var
0:
0> level-1-var
Error: symbol not bound: level-1-var
>
```

Our solution to this inconvenience is to simply share the global environment among all levels. The second drawback is that the file containing the interpreter definition must be loaded each time a new level is created. Since every level results in an order of magnitude slowdown, the time that it takes to create levels quickly becomes unbearably long, and having to wait inhibits experimentation. Sharing the global environment between levels makes it unnecessary to load the interpreter definition each time a new level is created.

Let's look more closely at the environment structure in a tower. Consider the previous transcript. The first line

```
> (load "simple.ss")
```

loads the procedure definitions for the interpreter into the Scheme environment. Then, the global environment of the interpreter is initialized to have bindings for the primitive procedures.

```
> (initialize-global-env)
```

In order to run a level-1 `openloop`, the level-0 interpreter's global environment must also include the definitions for the interpreter. This is accomplished by reloading the same file into the interpreter's global environment.

```
> (loadfile "simple.ss")
```

Then we start the interpreter's read-eval-print loop.

```
> (openloop "0> " "0: ")
0>
```

Although we could now open another level, it would fail because its global environment `global-env` is not bound in the level-0 interpreter's environment.

```
0> (openloop "1> " "1: ")
1> (+ 1 2)
Error: symbol not bound: global-env
>
```

The level-0 interpreter was attempting to run level-1 interpreter code that contained a reference to `global-env`. It was the level-0 interpreter that determined that `global-env` had not been bound, and thus exited to Scheme.

We can share the global environment of the interpreter among levels by including a binding for `global-env` in the level-0 interpreter's global environment. This binding must reference the interpreter's global environment so that all modifications are shared among all levels. Thus the global environment of the interpreter becomes a self-referential, circular structure. The following procedure initializes the interpreter's global environment so that it is shared among all levels, and starts a read-eval-print loop.

```
(set! boot-tower
  (lambda ()
    (initialize-global-env)
    (loadfile this-file-name)
    (set-cdr! global-env
      (cons (cons 'global-env global-env)
            (cdr global-env)))
    (openloop "0> " "0: ")))
```

With an initialized, self-referential, global environment, it is no longer necessary to run `initialize-global-env` and `loadfile` each time a new level is created. The following transcript creates a tower with three levels.

```
> (load "simple.ss")
> (boot-tower)
0> (openloop "1> " "1: ")
1> (openloop "2> " "2: ")
2>
```

Each successive `openloop` still results in an order of magnitude slowdown, but simply starting a new level is much faster than the method of previous sections.

We also have the capability to call `evaluate` or any other implementation procedure:

```
2> (evaluate '(+ 1 x) '((x . 3))
     (lambda (v) (* v 11)))
2: 44
2>
```

5.2 Reifying Procedures

We now consider adding reifying procedures to the interpreter. Reifying procedures are the mechanism that 3-LISP, Brown, and Blond provide to enable user code to run at the level of the interpreter with access to the interpreter's current expression, environment, and continuation.

5.2.1 Overview

In $\mathcal{I}_R$, reifying procedures are created by applying the procedure `compound-to-reifier` to a three-argument compound procedure. The result is a three-argument reifying procedure. When the operator of a combination evaluates to a reifying procedure, the body of the reifying procedure is run as if it were code of the interpreter evaluating the combination; the body is run in an extension of the reifying procedure's lexical environment where the first formal parameter is bound to the list of unevaluated operands of the combination, the second formal parameter is bound to the environment argument of the interpreter evaluating the combination, and the third formal parameter is bound to the continuation argument of the interpreter evaluating the combination. To illustrate, a transcript follows showing the definition and use of a reifying procedure that aborts the current level, returning its unevaluated first argument.

```
2> (set! quit
     (compound-to-reifier
       (lambda (e r k) (1st e))))
2:
2> (quit foo)
1: foo
1> (quit bar)
0: bar
0>
```

Interesting reifying procedures typically call `evaluate` in their bodies. Following is the definition of a special form (`when` *test exp*) that evaluates *exp* if and only if *test* evaluates to true.

```
2> (set! when
     (compound-to-reifier
       (lambda (e r k)
         (evaluate (1st e) r
           (lambda (v)
             (if v
                 (evaluate (2nd e) r k)
                 (k (void))))))))
2:
2> (when (= 1 2) "Shouldn't happen")
2:
2>
```

It is useful to define a special form `alpha` for abstracting reifiers. With `alpha`, `quit` could be defined by

```
(set! quit (alpha (e r k) (1st e)))
```

The body of an `alpha` is closed in the environment of its definition. The definition of `alpha` uses `make-reifier`, which, like `make-compound`, takes a list of formal parameters, a procedure body, and an environment, and directly creates the associated reifying procedure. We define `alpha` as follows:

```
(set! alpha
  (compound-to-reifier
    (lambda (e r k)
      (k (make-reifier (1st e) (cdr e) r)))))
```

5.2.2 Implementation

Reifying procedures are represented like compound procedures, except that the tag is `'reifier` instead of `'compound`. Converting between reifying procedures and compound procedures is then simply a matter of changing tags. Following are the operations on reifying procedures.

```
(set! make-reifier
  (lambda (formals body r)
    (list 'reifier formals body r)))

(set! reifier-to-compound
  (lambda (reifier)
    (cons 'compound (cdr reifier))))

(set! compound-to-reifier
  (lambda (compound)
    (cons 'reifier (cdr compound))))
```

All that remains is to handle the evaluation of combinations where the operator evaluates to a reifying procedure. Suppose that we have a combination (exp_0 exp_1 ...) where exp_0 evaluates to a reifying procedure bound to `proc`. Let *ext* be the extension of the lexical environment of `proc` where the formal parameters of `proc` are bound to the list of unevaluated operands of the combination, the environment of the interpreter evaluating the combination, and the continuation of the interpreter evaluating the combination, respectively. We want to run the body of `proc` at the level of interpreter code in the environment *ext*. This is the same as running the body of the value of

```
(reifier-to-compound proc)
```

at the level of interpreter code in the environment *ext*. By the *basic observation*, this is the same as running

```
((reifier-to-compound proc) (operands-part e) r k)
```

at the level of interpreter code. Thus, the following modification to `evaluate-combination`, indicated by semicolons, runs the body of a reifier at the proper level.

```
(set! evaluate-combination
  (lambda (e r k)
    (evaluate (operator-part e) r
      (lambda (proc)
        (if (reifier? proc)                          ;
          ((reifier-to-compound proc)                ;
            (operands-part e) r k)                   ;
          (evaluate-operands (operands-part e) r
            (lambda (args)
              (apply-procedure proc args k))))))))
```

This completes the description of the changes made to $\mathcal{I}$ in order to obtain $\mathcal{I}_R$. The full definition of $\mathcal{I}_R$ appears in the appendix.

The expression

```
((reifier-to-compound proc) (operands-part e) r k)
```

is the essence of our reflective tower model. It is essentially identical to the corresponding expression

```
(↓(de-reflect proc!) args env cont)
```

in the meta-circular definition of 3-Lisp [10]. The 3-Lisp procedure `de-reflect` performs the same function as the $\mathcal{I}_R$ procedure `reifier-to-compound`: it converts the closure for a reifier into a closure for a compound procedure. The only significant difference between the two expressions is the use of the meta-structural operator ↓ in the 3-Lisp expression so that the operator of the combination is a program (designating a function) rather than a structure (designating a closure).

5.3 Observations

The reflective interpreter $\mathcal{I}_R$ was obtained by making a few simple modifications to $\mathcal{I}$. The first modification created a self-referential global environment so that the interpreters in a tower could share the global environment. Then a datatype for reifying procedures was defined and a case was added to `evaluate-combination` so that it could handle applications of reifiers.

An error from the underlying Scheme system usually results if code run by level-0 attempts to apply a reifier:

```
0> ((compound-to-reifier
      (lambda (e r k) "Wow!")))
Error: attempt to apply non-procedure
       (compound (e r k) ("Wow!") ()).
>
```

The error is due to running the expression

```
((reifier-to-compound proc) (operands-part e) r k)
```

as level-0 $\mathcal{I}_R$ interpreter code. The level-0 $\mathcal{I}_R$ is run by the underlying Scheme system, which does not understand the representation of $\mathcal{I}_R$ procedures.

The following illustrates how it is possible to climb out of the finite tower and into Scheme, resulting in an error:

```
0> (openloop "1> " "1: ")
1> (openloop "2> " "2: ")
2> (set! climb
     (compound-to-reifier
       (lambda (e r k)
         (evaluate (1st e) r
           (lambda (v)
             (if (= v 0)
               (k v)
               (climb (- v 1))))))))
2:
2> (climb 0)
2: 0
2> (climb 1)
1: 0
1> (openloop "2> " "2: ")
2> (climb 2)
Error: attempt to apply non-procedure
       (compound (e r k) ((evaluate ...)) ()).
>
```

Due to the meta-circularity of $\mathcal{I}_R$, user programs have full access to the implementation and structures of $\mathcal{I}_R$. This is illustrated in the following transcript.

```
0> (openloop "1> " "1: ")
1> (openloop "2> " "2: ")
2> ((compound-to-reifier
      (lambda (e r k)
        (list e r k)))
    (* 3 5 ))
1: (((* 3 5))
    ()
    (compound (v)
      ((display write-prompt)
        (if (eq? v (void))
          "Nothing will be displayed"
          (write v))
        (newline)
        (openloop read-prompt write-prompt))
      ((write-prompt . "2: ")
       (read-prompt . "2> "))))
1> evaluate-abstraction
1: (compound (e r k)
     ((k (make-compound
           (formals-part e)
           (body-part e)
           r)))
     ())
1>
```

6 Comparison with Related Work

3-Lisp, Brown, Blond, and $\mathcal{I}_R$ are all closely related. All are interpreters based on the reflective tower architecture described in section 2. $\mathcal{I}_R$ models a finite reflective tower, whereas the others model an infinite reflective tower. All four languages are lexically scoped, higher order, applicative order dialects of Lisp. We do not include Stepper[1] because its reflective architecture differs substantially from that considered here.

3-Lisp differs in significant ways from other Lisp dialects. Before reflection is introduced, a semantically rationalized Lisp is defined. New internal structures are introduced in order to uniquely represent semantic concepts (e.g., combinations are not represented by list structure) and syntax is assigned to uniquely represent each type of internal structure. This results in an alignment of syntactic, structural, and semantic concepts. Quotation is a structural primitive in 3-Lisp that is used to designate internal structures. Levels of designation can only be removed or added explicitly via `up` and `down`. Evaluation is replaced by normalization, which takes an expression into a co-designating structure in normal form (e.g., `'2` normalizes to `'2`, not `2`). The result is that semantical levels may only be crossed with the explicit use of `up` and `down`. Reflection in 3-Lisp is then described as in section 2.

Meta-circular definitions of 3-Lisp are given in [9, 10, 11]. The meta-circular definition uses 3-Lisp to define 3-Lisp and hence is not well-founded. The definition of $\mathcal{I}_R$ is similar to the meta-circular definition of 3-Lisp. Level shifting and meta-circular access are implicit in both definitions, and the code for applying reifiers is essentially identical. The definition of $\mathcal{I}_R$ differs in the following respects:

- It models a finite tower, rather than an infinite tower.
- It is well-founded and can be run in Scheme. The meta-circular definition of 3-Lisp can only be run by a 3-Lisp implementation.
- It does not require an understanding of 3-Lisp's reconstruction of evaluation and quotation.

The non-reflective implementations [9, 4] of 3-Lisp are extremely operational. In order to operate efficiently, level shifting in the tower is explicitly modeled with a stack of continuations corresponding to the active levels of the tower. Full access to the interpreter, implied by meta-circularity, is explicitly simulated.

Brown and Blond are reflective dialects of Scheme, both of which are implemented in Scheme. Both are essentially executable denotational definitions of the non-meta-circular aspects of the reflective tower architecture. Level shifting in the tower is explicitly modeled by adding a new context argument, called the meta-continuation, to the semantic valuation functions.

As $\mathcal{I}_R$ demonstrates, meta-circularity implies that user programs have the same access privileges as the interpreter itself. Thus, the procedures that implement a meta-circular interpreter may be used, inspected, and modified by a user program. Environments and continuations can be similarly accessed by a user program. $\mathcal{I}_R$ user programs have the same access privileges as $\mathcal{I}_R$ itself. The implementation of 3-Lisp faithfully simulates its corresponding meta-circular definition, but protects itself by not allowing modification of the standard 3-Lisp system. Nevertheless, 3-Lisp user programs have full inspection privileges.

In Blond and Brown, the internal structure of the interpreter is largely hidden from user programs. Brown and Blond make no attempt to model the meta-circular properties of the reflective tower, but rather focus on the denotational semantics of shifting between levels of the tower. Both Brown and Blond rely upon Scheme closures to represent user-defined procedures, interpreter procedures and interpreter continuations. Brown also represents the environment as a closure. The only sanctioned operation that either the Blond or Brown interpreters or their user programs can perform on Scheme closures is application.

In 3-Lisp, Brown, and $\mathcal{I}_R$, reifiers close in the environment of their definition. In Blond, reifiers close in either the environment of the level above their definition or in the environment above their level of application. Reifiers of the first kind are defined via `gamma` abstractions, and those of the second kind are defined via `delta` abstractions.

In 3-Lisp, Brown, and $\mathcal{I}_R$, if the application of a reified continuation (i.e., interpreter continuation) returns, then execution is resumed at the point following the application. This procedure-like behavior of reified continuations has been termed "pushy" [5]. In $\mathcal{I}_R$ this pushy behavior is a natural consequence of actually running a literal tower of interpreters. Brown and 3-Lisp simulate the pushy behavior that is a consequence of the reflective tower architecture. The designers of Blond argue for "jumpy" reified continuations, which behave more like a `goto` when they are applied. The jumpy behavior is obtained by forgetting the continuation in effect when a reified continuation is applied.

Blond has a separate environment on each level and a global environment that is common to all levels. Brown and $\mathcal{I}_R$ have only a global environment that is common to all levels. Both approaches have been used in different 3-Lisp implementations.

7 Conclusion

An executable reflective interpreter has been defined by making a few minor modifications to an ordinary interpreter for a dialect of Scheme. The reflective interpreter literally models a finite reflective tower. This makes it too slow for practical use, but its simplicity makes it ideal for understanding and experimenting. Once the literal model of a finite reflective tower is understood, the behavior of an infinite reflective tower is easily understood as the limiting behavior of a finite reflective tower as its height approaches infinity.

Our reflective interpreter illustrates how switching levels in the tower involves treating data as code and vice-versa. For those who are uncomfortable with this sort of semantical level crossing, it serves as additional motivation for understanding the reconstruction of evaluation and quotation in 3-Lisp. [9, 10].

Acknowledgements

We are grateful to John Simmons for using the system and providing much helpful feedback. We would like to thank Jim des Rivières, Steve Greenbaum, Evan Kirshenbaum, Daniel Kuokka, and Julia Lawall for their comments on earlier drafts of this paper. The first author is grateful to Hewlett-Packard Laboratories for their support and the use of their facilities in completing this paper.

References

[1] Bawden, A., Reification without evaluation, *Conference Record of the 1988 ACM Symposium on LISP and Functional Programming*, Snowbird, Utah, July 1988, 342–351.

[2] Danvy, O., and Malmkjær, K., Intensions and Extensions in a Reflective Tower, *Conference Record of the 1988 ACM Symposium on LISP and Functional Programming*, Snowbird, Utah, July 1988, 327–341.

[3] Danvy, O., and Malmkjær, K., A Blond Primer, DIKU Rapport, DIKU, Computer Science Dept., University of Copenhagen, Copenhagen, Denmark, September 1988.

[4] des Rivières, J., and Smith, B.C., The Implementation of Procedurally Reflective Languages, *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, August 1984, 331–347.

[5] des Rivières, J., Control-Related Meta-Level Facilities in LISP (Extended Abstract), *Meta-Level Architectures and Reflection*, Patti Maes and Daniele Nardi (eds.), North-Holland, 1988, 101–110.

[6] Friedman, D.P., and Wand, M., Reification: Reflection without Metaphysics, *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, August 1984, 348–355.

[7] Maes, P., and Nardi, D. (eds.), *Meta-Level Architectures and Reflection*, North-Holland, 1988.

[8] Rees, J., and Clinger, W., eds. Revised^3 Report on the Algorithmic Language Scheme, *SIGPLAN Notices 21*,12 (December, 1986), 37–79.

[9] Smith, B.C., *Reflection and Semantics in a Procedural Language*, MIT-LCS-TR-272, Mass. Inst. of Tech., Cambridge, MA, January, 1982.

[10] Smith, B.C., Reflection and Semantics in Lisp, *Conf. Rec. 11th ACM Symp. on Principles of Programming Languages*, 1984, 23–35.

[11] Smith, B.C., and des Rivières, J., *Interim 3-LISP Reference Manual*, Intelligent Systems Laboratory, Xerox PARC, Palo Alto, California, 1984.

[12] Wand, M., and Friedman, D.P., The Mystery of the Tower Revealed: a non-Reflective Description of the Reflective Tower, *Lisp and Symbolic Computation*, vol. 1, no. 1, June 1988, 11–38.

Appendix: The Complete Reflective Interpreter

Following is a listing of a Scheme source file, which we have named `simple.ss`, that defines the interpreter presented in section 5. After loading the file, the interpreter is started by typing `(boot-tower)`.

```
;;;
;;; A Simple Reflective Interpreter
;;;
;;;
;;; In some Scheme systems it may be necessary to
;;; first load a separate file that initializes
;;; every global variable in this file to some
;;; arbitrary value using DEFINE.  It may also be
;;; necessary to include the following definition
;;; in that file:
;;;
;;; (define void
;;;   (let ((g (cons '* '*)))
;;;     (lambda () g)))
;;;

(set! evaluate
  (lambda (e r k)
    ((if (constant? e)
       evaluate-constant
       (if (variable? e)
         evaluate-variable
         (if (if? e)
           evaluate-if
           (if (assignment? e)
             evaluate-assignment
             (if (abstraction? e)
               evaluate-abstraction
               evaluate-combination)))))
      e r k)))

(set! evaluate-constant
  (lambda (e r k)
    (k (constant-part e))))

(set! evaluate-variable
  (lambda (e r k)
    (get-pair e r
      (lambda (success-pair)
        (k (cdr success-pair)))
      (lambda ()
        (wrong "symbol not bound: " e)))))

(set! wrong
  (lambda (message object)
    (display "Error: ")
    (display message)
    (display object)
    (newline)))

(set! evaluate-if
  (lambda (e r k)
    (evaluate (test-part e) r
      (lambda (v)
        (if v
          (evaluate (then-part e) r k)
          (evaluate (else-part e) r k))))))

(set! evaluate-assignment
  (lambda (e r k)
    (evaluate (value-part e) r
      (lambda (v)
        (get-pair (id-part e) r
          (lambda (success-pair)
            (set-cdr! success-pair v)
            (k (void)))
          (lambda ()
            (set-cdr! global-env
              (cons (cons (id-part e) v)
                (cdr global-env)))
            (k (void))))))))

(set! evaluate-abstraction
  (lambda (e r k)
    (k (make-compound
         (formals-part e) (body-part e) r))))

(set! evaluate-combination
  (lambda (e r k)
    (evaluate (operator-part e) r
      (lambda (proc)
        (if (reifier? proc)
          ((reifier-to-compound proc)
            (operands-part e) r k)
          (evaluate-operands (operands-part e) r
            (lambda (args)
              (apply-procedure proc args k))))))))

(set! evaluate-operands
  (lambda (operands r k)
    (if (null? operands)
      (k '())
      (evaluate (car operands) r
        (lambda (v)
          (evaluate-operands (cdr operands) r
            (lambda (w)
              (k (cons v w)))))))))

(set! evaluate-sequence
  (lambda (body r k)
    (if (null? (cdr body))
      (evaluate (car body) r k)
      (evaluate (car body) r
        (lambda (v)
          (evaluate-sequence (cdr body) r k))))))

(set! apply-procedure
  (lambda (proc args k)
    (if (compound? proc)
      (evaluate-sequence
        (procedure-body proc)
        (extend
          (procedure-environment proc)
          (procedure-parameters proc)
          args)
        k)
      (k (apply-primitive
           (procedure-name proc) args)))))
```

```
(set! apply-primitive
  (lambda (name args)
    (if (eq? name 'car)
        (car (1st args))
    (if (eq? name 'cdr)
        (cdr (1st args))
    (if (eq? name 'cons)
        (cons (1st args) (2nd args))
    (if (eq? name 'set-car!)
        (set-car! (1st args) (2nd args))
    (if (eq? name 'set-cdr!)
        (set-cdr! (1st args) (2nd args))
    (if (eq? name 'assq)
        (assq (1st args) (2nd args))
    (if (eq? name 'memq)
        (memq (1st args) (2nd args))
    (if (eq? name 'null?)
        (null? (1st args))
    (if (eq? name '=)
        (= (1st args) (2nd args))
    (if (eq? name 'eq?)
        (eq? (1st args) (2nd args))
    (if (eq? name 'newline)
        (newline)
    (if (eq? name 'write)
        (write (1st args))
    (if (eq? name 'display)
        (display (1st args))
    (if (eq? name 'read)
        (if (null? args) (read) (read (1st args)))
    (if (eq? name '+)
        (+ (1st args) (2nd args))
    (if (eq? name '-)
        (- (1st args) (2nd args))
    (if (eq? name '*)
        (* (1st args) (2nd args))
    (if (eq? name 'symbol?)
        (symbol? (1st args))
    (if (eq? name 'list)
        args
    (if (eq? name 'pair?)
        (pair? (1st args))
    (if (eq? name 'eof-object?)
        (eof-object? (1st args))
    (if (eq? name 'close-input-port)
        (close-input-port (1st args))
    (if (eq? name 'open-input-file)
        (open-input-file (1st args))
    (if (eq? name 'void)
        (void)
        "Shouldn't Happen!"))))))))))))))))))))))))))

;;;
;;; Environments.
;;;

(set! extend
  (lambda (r ids vals)
    (if (null? ids)
      r
      (extend
        (cons (cons (car ids) (car vals)) r)
        (cdr ids)
        (cdr vals)))))

(set! get-pair
  (lambda (id r success failure)
    (find-pair id r
      success
      (lambda ()
        (find-pair
          id global-env success failure)))))

(set! find-pair
  (lambda (elt alist success failure)
    ((lambda (assq-result)
       (if assq-result
           (success assq-result)
           (failure)))
     (assq elt alist))))

(set! empty-env '())

;;;
;;; List utilities.
;;;

(set! 1st (lambda (l) (car l)))
(set! 2nd (lambda (l) (car (cdr l))))
(set! 3rd (lambda (l) (car (cdr (cdr l)))))
(set! 4th (lambda (l) (car (cdr (cdr (cdr l))))))

(set! test-tag
  (lambda (tag)
    (lambda (e)
      (if (pair? e) (eq? (car e) tag) #f))))

;;;
;;; Procedures.
;;;

(set! make-compound
  (lambda (formals body r)
    (list 'compound formals body r)))

(set! compound? (test-tag 'compound))

(set! make-primitive
  (lambda (op)
    (list 'primitive op)))

(set! primitive? (test-tag 'primitive))

(set! primitive-identifiers
  (lambda ()
    '(car cdr cons set-car! set-cdr! assq memq
       null? = eq? newline write display read
       + - * symbol? list pair? eof-object?
       close-input-port open-input-file void)))

(set! make-reifier
  (lambda (formals body r)
    (list 'reifier formals body r)))

(set! reifier-to-compound
  (lambda (reifier)
    (cons 'compound (cdr reifier))))
```

```
(set! compound-to-reifier
  (lambda (compound)
    (cons 'reifier (cdr compound))))

(set! reifier? (test-tag 'reifier))

(set! procedure-parameters 2nd)
(set! procedure-body 3rd)
(set! procedure-environment 4th)
(set! procedure-name 2nd)

;;;
;;; Syntax.
;;;

(set! variable? symbol?)
(set! if? (test-tag 'if))
(set! assignment? (test-tag 'set!))
(set! abstraction? (test-tag 'lambda))
(set! quote? (test-tag 'quote))

(set! constant?
  (lambda (e)
    (if (pair? e) (quote? e)
      (if (symbol? e) #f #t))))

(set! constant-part
  (lambda (e) (if (quote? e) (2nd e) e)))

(set! test-part 2nd)
(set! then-part 3rd)
(set! else-part 4th)

(set! id-part 2nd)
(set! value-part 3rd)

(set! formals-part 2nd)
(set! body-part (lambda (e) (cdr (cdr e))))

(set! operator-part 1st)
(set! operands-part cdr)

;;;
;;; Read-Eval-Print Loop and Loadfile
;;;

(set! openloop
  (lambda (read-prompt write-prompt)
    (display read-prompt)
    (evaluate (read) empty-env
      (lambda (v)
        (display write-prompt)
        (if (eq? v (void))
            "Nothing will be displayed"
            (write v))
        (newline)
        (openloop read-prompt write-prompt)))))

(set! loadfile
  (lambda (file)
    ((lambda(port)
       ((lambda (loop)
          (set! loop
            (lambda (v)
              (if (eof-object? v)
                (close-input-port port)
                (evaluate v empty-env
                  (lambda (ignore)
                    (loop (read port)))))))
          (loop (read port)))
        '*))
     (open-input-file file))))

;;;
;;; The booting process
;;;

(set! mapper
  (lambda (f l)
    (if (null? l)
      '()
      (cons (f (car l)) (mapper f (cdr l))))))

(set! initialize-global-env
  (lambda ()
    (set! global-env
      (extend
        empty-env
        (primitive-identifiers)
        (mapper make-primitive
          (primitive-identifiers))))))

(set! boot-flat
  (lambda ()
    (initialize-global-env)
    (openloop "0< " "0> ")))

(set! boot-tower
  (lambda ()
    (initialize-global-env)
    (loadfile this-file-name)
    (set-cdr! global-env
      (cons (cons 'global-env global-env)
            (cdr global-env)))
    (openloop "0> " "0: ")))

(set! this-file-name "simple.ss")
```