The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower

By

Mitchell Wand
Northeastern University
Boston, Massachusetts 02115

and

Daniel P. Friedman
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 196

The Mystery of the Tower Revealed:
A Non-Reflective Description of the Reflective Tower

by

Mitchell Wand, Northeastern University
and Daniel P. Friedman, Indiana University

May, 1986

This material is based on work supported by the National Science Foundation under grant numbers MCS 83-04567, MCS 83-03325 and DCR 86-05218.

This report to appear in the Proceedings of the 1986 ACM Symposium on LISP and Functional Programming.

The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower

Mitchell Wand, Northeastern University
Daniel P. Friedman, Indiana University

1. Abstract

In an important series of papers [Smith 82, 84], Brian Smith has discussed the nature of programs which are allowed to have knowledge of their text and of the context in which they are executed. He called this kind of knowledge *reflection*. Smith proposed a programming language, called 3-LISP, which embodied such self-knowledge in the domain of meta-circular interpreters. Every 3-LISP program is interpreted by a metacircular interpreter, also written in 3-LISP. This gives rise to a picture of an infinite tower of meta-circular interpreters, each being interpreted by the one above it. Such a metaphor poses a serious challenge for conventional modes of understanding of programming languages.

In our earlier work on reflection [Friedman & Wand 84], we showed how a useful species of reflection could be modelled without the use of towers. During the question period following that presentation, Smith challenged us to extend our techniques to give a model of towers as well. In this paper, we meet this challenge by giving a semantic account of the reflective tower. This account is self-contained in the sense that it does not employ reflection to explain reflection.

This Material is based on work supported by the National Science Foundation under grant numbers MCS 8303325, MCS 8304567, and DCR 8605218.

Authors' addresses: Mitchell Wand, College of Computer Science, Northeastern University, 360 Huntington Avenue #161CN, Boston, MA 02115. Daniel P. Friedman, Computer Science Department, Indiana University, Lindley Hall 101, Bloomington, IN 47405

To appear in the 1986 ACM Symposium on LISP and Functional Programming.

2. Modelling Reflection

Let us first consider how a conventional denotational semantics models the context in which a computation takes place. In a conventional language, an expression is evaluated in a context which includes several parts:

1. An *environment* that describes the bindings of identifiers, which, depending on the language, might be values or locations.
2. A *continuation* that describes the control context. This is typically modelled by a function whose job it is to receive the answer from the current expression and then finish the entire calculation.
3. A *store* that describes the "global state" of the computation, including the contents of locations and the state of the input-output system. In this paper, we do not deal with the store part of the context.

These pieces of context are taken into account by passing them as arguments to the valuation or interpreter. Thus the type of the interpreter or valuation is

$$\mathcal{E} : \langle Exp \rangle \to Env \to K \to A = \lambda e \rho \kappa. \ \ldots$$

where A is some domain of answers. Thus we can think of $\mathcal{E}$ (once we have written it out) as defining an interpreter which manipulates three registers, e, ρ, and κ.

In [Friedman & Wand 84] we showed how Smith's concept of reflection can be decomposed into two processes, which we called *reification* and *reflection*. We used the term reification for the process by which the contents of the interpreter registers, e, ρ, and κ, are passed to the program itself, suitably packaged (or *reified*) so the program can manipulate them. We think of this process as converting program into data. Conversely, reflection is the process by which program values for an expression, an environment, and a continuation are re-installed as the values of the interpreter registers. This process may be thought of as turning data into program.

Following Smith, we built reification into our language by using a special class of procedure (called, albeit confusingly, *reflective procedures*). Such a procedure, when called, bound the reified contents of the interpreter registers to the formal parameters and then executed the body. This mechanism generalized the Lisp Fexpr mechanism. Reflection was built in using a function called `meaning` which took three arguments and installed them as the values of the interpreter registers (thus generalizing Lisp's `eval`). With this model, we were able to generalize the conventional treatment of special forms by making them first-class citizens. All this was done without having to introduce the concept of reflective towers.

Having shown that the concept of reification was independent of the idea of the infinite tower, we then were led to consider the question of providing a reasonable model of the tower itself. Reflective models (as in [Smith 84]) were unsatisfactory for foundational reasons: they depended on an understanding of reflective towers in the defining language, when that was precisely the feature we hoped to explain. The only non-reflective models ([Smith 82, Chapter 5; Smith & des Rivières 84b]) were extremely operational. Indeed, it was not clear whether the techniques of denotational semantics were adequate to describe the tower.

In the next section, we shall show how denotational semantics can be used to describe a tower of computations.

3. Modelling the Tower

3-LISP adds to the reflective structure a serious commitment to the idea of meta-circular interpreters. Every 3-LISP program is interpreted by a metacircular interpreter, also written in 3-LISP, which in turn is interpreted by a metacircular interpreter above it, and so on. This leads to an infinite tower of interpreters, each manipulating an expression, an environment, and a continuation. Each interpreter runs in a context consisting of the states of the interpreters above it.

This yields a slightly different picture of reification and reflection. When reflection occurs (by invocation of the function `meaning`), a new interpreter is spawned below the current one. When the lower interpreter exits, control returns to the interpreter which spawned it. When the lower interpreter invokes a reflective procedure, its registers are reified and passed to the body of the reflective procedure, as in our earlier model. In 3-LISP, however, the body of the reflective procedure is then executed *as if it were in the upper interpreter*. Thus, 3-LISP's reflective procedures, like Lisp's special forms, effectively add new lines to the interpreter.

Our standard treatment of contextual information gives a straightforward way of modelling this situation. We simply change the type of the semantic function so that it takes, in addition to the usual expression, environment, and continuation, a new piece of context information that we call a *metacontinuation*. Hence the type of $\mathcal{E}$ is now

$$\mathcal{E} : \langle Exp \rangle \rightarrow Env \rightarrow K \rightarrow MK \rightarrow A$$

The problem is characterizing the domain MK. A metacontinuation represents the state of the upper interpreter (and by implication that of the tower above it) waiting for a result from the lower interpreter. Thus MK will have the form

$$MK = R \rightarrow A$$

where R is the domain of interpreter results. Our next task is to determine the domain R. To do this, we will think primarily about what happens when the lower interpreter invokes a reflective procedure and returns to the next level.

When the lower interpreter executes a reflective procedure, the body of the procedure is run at the place the meaning function was called in the upper interpreter. What do we mean here by "body?" Clearly we mean an object built from an expression (the body of the procedure) and an environment (built from the lexical environment of the procedure extended with the formal parameters bound to the actuals)—in other words, a thunk. Looking at the functionality of $\mathcal{E}$, we see that combining an expression and an environment gives us an object of type $K \rightarrow MK \rightarrow A$. When a reflective procedure is invoked, the appropriate thunk is built and passed to the metacontinuation. Thus the domain of metacontinuations should be

$$MK = [K \rightarrow MK \rightarrow A] \rightarrow A$$

How is this thunk built? It is an object built from the body of the procedure and an environment consisting of the lexical environment of the procedure with the formal parameters bound to the actuals. In the case of a reflective procedure, the actuals are the e, ρ, and κ, suitably packaged (or *reified*) so that the body can use them.

What do we mean here by "place?" We mean the continuation (and metacontinuation) in force at the time `meaning` was called. When a new interpreter is spawned at continuation κ and metacontinuation μ, we expect it to return a thunk θ which will be run on continuation κ and metacontinuation μ. Thus, the lower interpreter should be run with metacontinuation

$$\lambda\theta.\theta\kappa\mu$$

That is, the operation of building this new metacontinuation is

$$\textit{meta-cons} = \lambda\kappa\mu\theta.\theta\kappa\mu$$

In Scheme, this might be written as:

```
(define meta-cons
  (lambda (k)
    (lambda (mk)
      (lambda (theta)
        ((theta k) mk)))))
```

This combinator is just Church's pairing combinator [Barendregt 81, p. 129], so it is not far wrong to think of a metacontinuation as a list of interpreters (or continuations). Writing this functionally, however, allows us to form an infinite tower using the standard fixpoint combinator:

$$\mu_\infty = Y(\lambda\mu.\mathit{meta\text{-}cons}\,\kappa_0\,\mu)$$

where κ_0 is the initial continuation used to initialize each interpreter in the tower.

In Scheme, this might be written as

```
(define tower
  (letrec
    ([loop
      (lambda (n)
        (lambda (theta)
          ((theta (R-E-P n)) (loop (add1 n)))))])
    (loop 0)))
```

where (R-E-P n) generates the initial continuation (a read-eval-print loop) for the interpreter at level n. Thus each interpreter begins with a continuation which is a read-eval-print loop.

What about termination of the lower interpreter? Let us imagine that we want to terminate the lower interpreter with value v. To do this, we must pass v to the continuation κ waiting in the upper interpreter. Thus we must pass to the metacontinuation a thunk which, given κ, passes v to it. This can be done by invoking a continuation initk defined as follows:

```
(define initk
  (lambda (v)
    (lambda (mk)
      (mk (lambda (k) (k v))))))
```

4. Up and Down the Tower

In this section we will give a glimpse of some of the programming techniques that are made possible in the tower, and try to compare these with the towerless reification of [Friedman & Wand 84]. Our understanding of this powerful tool is still sketchy, but we will attempt to share what we do understand.

We call this language *Brown*. Its surface syntax is familiar. It has identifiers, abstractions (for which we use the notation (lambda (id ...) body), and combinations of any number of arguments. This much of the language behaves like the conventional applicative-order language.

Reflection is built into the language through two primitives, meaning and make-reifier. meaning takes three arguments: an expression, an environment, and a continuation, and starts a new interpreter with these three values as the initial contents of the registers. make-reifier takes a 3-argument abstraction and turns it into a reflective procedure that, when called, reifies the registers e, ρ, and κ into Brown values, creates a suitable thunk, and passes it to the metacontinuation. Such a reflective procedure behaves as if its body were being executed by the interpreter. Consider for example:

```
(make-reifier
  (lambda (e r k)
    (meaning (car (cdr e)) r
      (lambda (v)
        (k (set-cell! (r (car e)) v))))))
```

This builds a reifier, which, when invoked on an expression consisting of two arguments, does the following: First, the second argument is evaluated, yielding a value v. Then the environment is queried using the first argument (unevaluated) to supply a cell, the resulting cell is modified (using the primitive set-cell!) and the resulting value sent to the continuation k. This would be an appropriate reifier to be bound to the name set!. We can do this by using it on itself, in the following code (to be executed in Brown!):

```
((lambda (setter)
   (setter set! setter))
 (make-reifier
   (lambda (e r k)
     (meaning (car (cdr e)) r
       (lambda (v)
         (k (set-cell! (r (car e)) v)))))))
```

Once we have defined set!, we can do all further definition inside the language. Hence in this paper, all definitions performed with set! are in Brown, and all definitions performed with define are in Scheme. So, for example, we can execute the following in Brown:

```
(set! if
  (make-reifier
    (lambda (e r k)
      (meaning (car e) r
        (lambda (v)
          (meaning
            (ef v
                (car (cdr e))
                (car (cdr (cdr e))))
            r k))))))
```

This code defines if so that in (if exp0 exp1 exp2), exp0 is evaluated first, in a continuation which evaluates either exp1 or exp2, depending on the value returned by exp0. The code uses the "extensional if" function ef, which takes a boolean and two values and returns one of the two values. Unlike if, ef is purely functional; it may be defined in Scheme as:

```
(define ef
  (lambda (bool x y)
    (if bool x y)))
```

We may now begin exploring the vagaries of the tower world. We begin with quote, which may be defined as:

```
(set! quote
  (make-reifier
    (lambda (e r k) (k (car e)))))
```

This function, when invoked, takes its first argument and passes it unevaluated to the call-time continuation. This is, of course, just what quote is supposed to do. Now consider

```
(set! jump
  (make-reifier
   (lambda (e r k) (car e))))
```

This function, when invoked, merely returns its first argument unevaluated. But returns it to what? In a towerless world, this would simply terminate the computation. With the tower, however, the effect is to terminate the current interpreter and return this value to the continuation waiting in the upper interpreter. Thus:

```
>>> (boot-tower)      ; start the tower

0:: starting-up
0-> (jump foo)
1:: foo
1-> (jump bar)
2:: bar
2-> (jump baz)
3:: baz
3->
```

and so on. A function which evaluates its argument and then exits might be written as follows:

```
(set! exit
  (lambda (x)
    ((make-reifier
       (lambda (e r k) x)))))
```

When invoked, this function receives a value x. It then creates and immediately invokes a reifier that exits from the current interpreter with x as its value.

We can open up a new read-eval-print loop using openloop:

```
(set! openloop
  (lambda (prompt)
    ((readloop prompt) 'starting-up)))
```

Here readloop is a primitive function which takes a prompt and produces a Brown continuation (see Section 5.6 below). We invoke readloop to create the continuation and then invoke it with an arbitrary value, starting-up, which is printed as the first response of the readloop.

Thus we might get the following dialog:

```
0:: starting-up
0-> (exit 'foo)             ; exit from this reader
                            ; and go up the tower.
1:: foo                     ; here we are at level 1.
1-> (exit 'bar)             ; let's do it again.
2:: bar
2-> (exit 'baz)             ; and again.
3:: baz
3-> (openloop 'N)           ; now let's open up a
                            ; new loop under loop
                            ; number 3.  Prompts
                            ; are arbitrary.
N:: starting-up
N-> (exit 'bow)             ; now we'll go back to
                            ; the creator of this
                            ; loop,
3:: bow                     ; which is number 3, as
                            ; expected.
3->
```

Now we can define call/cc as follows:

```
(set! call/cc
  (lambda (f)
    ((make-reifier
       (lambda (e r k) (k (f k)))))))
```

This function receives a function, immediately reifies (as exit did above), and applies f to the continuation k. If the invocation of f returns normally, control should return to the continuation k. Thus

```
(call/cc (lambda (k) '3))
```

should be the same as '3. What happens, however, if k is invoked within f? In a towerless world, the invocation of a continuation is a "black hole": the current continuation is thrown away and the new one is installed in its place. In the tower model, things are not so simple. Consider the following example [J. des Rivières, private communication]:

```
0:: starting-up
0-> (call/cc
      (lambda (k)
        (cons (k '2) (k '3))))
0:: 2
```

Here k becomes bound to the level-0 readloop. Then (cons (k '2) (k '3)) is evaluated by the upper interpreter. When it invokes k on 2, it prints the 2 and continues with the level-0 readloop, remembering (via meta-cons) that the lower interpreter was invoked from inside the cons. Thus, when the lower interpreter terminates, the value it returns will be passed as the first argument to cons. The next step is to evaluate the second argument to cons, in this case (k '3). Again, since k is bound to the level-0 readloop, level 0 is started again. So, if we do an exit, we do not get to the level 1 readloop, but we immediately bounce down to level 0 again:

```
0-> (exit 'foo)
0:: 3                    ; instead of  1:: foo
```

If we cause the level-0 readloop to exit, its termination value becomes the value of (k '3). Level 1 then does the cons, and passes the value to k, which restarts the level-0 readloop (for the third time):

```
0-> (exit 'bar)
0:: (foo . bar)
0->
```

What would happen if we used a different variant of call/cc, closer to that analyzed in [Felleisen *et al.* 86]?

```
(set! new-call/cc
  (lambda (f)
((make-reifier (lambda (e r k) (f k)))))
```

This is similar to the previous version, except that it expects (f k) to terminate by invoking k. This will behave in exactly the same way as the previous example, except that when the cons terminates it sends its value to the level-1 readloop instead of re-invoking level 0, so that the last few lines would be:

```
0-> (exit 'bar)
1:: (foo . bar)
1->
```

Other bizarre things are possible. Consider

```
(set! strange
  (lambda ()
    (new-call/cc
      (lambda (k) (set! new-k k)))))
```

This is a function which, when invoked, sets a global variable new-k to the current readloop and then exits the current readloop. A subsequent invocation of new-k will jump back to the readloop from which strange was called. If that readloop is terminated (via exit or even via strange again) then control will return to the readloop from which new-k was called.

Clearly we have only begun to explore the possibilities inherent in the tower model.

5. The Model

In this section, we begin a commented tour of the model. We have expressed it in "pure" Scheme, without side-effects or call/cc, except for use in the interface between the implementation and the outside world. We believe that this is sufficiently close to denotational semantics to allow a relatively straightforward transcription. The model as presented here is also complete and testable. Most of the code is included in the text; a few help functions are left for an appendix.

5.1 Currying

Almost every function in the semantics is fully Curried. This allows us to delete extraneous arguments, as is typically done in semantic specifications. To make this easier, we begin with some syntactic extensions which allow us to proceed without fully parenthesizing all the applications and nested lambdas. We do this using the macro-declaration tool extend-syntax [Kohlbecker 86].

```
(extend-syntax (C)
  [(C m n) (m n)]
  [(C m n p ...) (C (m n) p ...)])
```

```
(extend-syntax (curry)
  [(curry (i) b ...) (lambda (i) b ...)]
  [(curry (i j ...) b ...)
   (lambda (i)
     (curry (j ...) b ...))])
```

With these, we can re-write `meta-cons` as follows:

```
(define meta-cons
  (curry (k mk theta)
    (C theta k mk)))
```

5.2 Denotations

The main function in the semantics is `denotation`, which branches on the syntactic type of an expression and then dispatches to one of three semantic functions:

```
(define denotation
  (lambda (e)
    (cond
      [(atom? e) (denotation-of-identifier e)]
      [(eq? (first e) 'lambda)
       (denotation-of-abstraction e)]
      [else (denotation-of-application e)])))
```

In keeping with the functionalities discussed above, each semantic function is of type

$$Exp \rightarrow Env \rightarrow K \rightarrow MK \rightarrow A$$

An expression is represented as a list structure in the usual way. An environment is represented as a function of two (curried) arguments: an identifier and a continuation waiting for the L-value associated with that identifier. A continuation or metacontinuation is represented as a function of one argument. Metacontinuations do not appear in the semantic functions, since (for the moment) we are modelling only a single interpreter. They will appear in some of the primitives, since it is through the primitives that reflection and reification occur. (This is analogous to the conventional presentation of denotational semantics, in which, for example, a store argument almost always appears in the definitions of the primitives, rather than in the main semantic equations. This is one way in which the equations may be made modular).

If the expression is an identifier, then the identifier is passed to the environment, along with a continuation to dereference the returned cell. By convention, a cell is returned even for an unbound identifier.

```
(define denotation-of-identifier
  (curry (e r k)
    (C r e
      (lambda (cell)
        (let ([v (deref-cell cell)])
          (if (eq? v 'UNASSIGNED)
              (wrong
                (list "Brown: unbound id " e))
              (k v)))))))
```

In order to accomodate reification, Brown uses call-by-text. A Brown function has functionality

$$BF = Exp* \rightarrow Env \rightarrow K \rightarrow MK \rightarrow A$$

It gets the text of the actual parameters, the call-time environment, and the call-time continuation and meta-continuation, and from this information computes an answer:

```
(define denotation-of-application
  (curry (e r k)
    (C denotation (first e) r
       (lambda (f) (C f (rest e) r k)))))
```

If the expression is an abstraction, we produce the usual procedure object—a function which accepts a sequence of values and then evaluates the body of the abstraction in a suitably extended environment—convert it to a call-by-text function using the auxiliary `F->BF`, and pass the result to the continuation:

```
(define denotation-of-abstraction
  (curry (e r k)
    (k (F->BF
         (lambda (v*)
           (C denotation (third e)
              (extend r (second e) v*)))))))
```

The function `F->BF` takes an element of F $(= V \rightarrow K \rightarrow MK \rightarrow A)$ and turns it into a Brown function which evaluates its actual parameters in the call-time environment and passes the list of results to the function:

```
(define F->BF
  (curry (fun e r k)
    (C Y (curry (eval-args e k)
           (if (null? e) (k '())
               (C denotation (first e) r
                  (lambda (v)
                    (C eval-args (rest e)
                       (lambda (w)
                         (k (cons v w))))))))
       e (curry (v* mk) (C fun v* k mk)))))
```

This code uses the applicative-order `Y` combinator (see Appendix).

5.3 Reification

We next turn to the reifying functions. These functions take objects from the underlying domains K and Env, and turn them into Brown functions which can be manipulated

[Friedman & Wand 84]. An environment is turned into a one-argument brown function which evaluates its argument and passes the result to the environment:

```
(define U->BF
  (curry (r1 e r k)
    (if (= (length e) 1)
        (C denotation (first e) r
           (lambda (v) (C r1 v k)))
        (wrong (list
                 "U->BF: wrong number of args "
                 e)))))
```

Continuations are treated similarly. Here k1 is the continuation to be converted, and e, r, and k are the Brown interpreter's registers at the point that k1 is invoked. Since a continuation is regarded as restarting a lower interpreter, we save the continuation k by putting it in the meta-continuation with meta-cons, as discussed in Section 3:

```
(define K->BF
  (curry (k1 e r k)
    (if (= (length e) 1)
        (lambda (mk)
          (C denotation (first e) r
             k1 (C meta-cons k mk)))
        (wrong (list
                 "K->BF: wrong number of args "
                 e)))))
```

Here is where the tower model begins to be radically different from the non-tower model. We have two continuations to deal with, but without the tower we have only one continuation register. The presence of the metacontinuation gives us a place to save the second continuation. In the corresponding function schemeK-to-brown in [Friedman & Wand 84], we simply threw away the continuation k corresponding to the point that (K->BF k1) was invoked.

5.4 Building Reflective Procedures

We need to write a function which takes a simple brown function and converts it into a reflective procedure: a brown function that reifies its arguments and passes the resulting thunk to the metacontinuation. A first try at this might be:

```
(define make-reifier
  (curry (bf e r k mk)
    (mk (bf
          (list e (U->BF r) (K->BF k))))))
```

where U->BF and K->BF reify environments and continuations, respectively.

This version does not quite work, however. The problem is that bf is a call-by-text function which takes a sequence of texts (the actual parameters), not a list of values.

How can we fool a call-by-text function like bf into taking values instead? We assume that bf is a simple abstraction, which will evaluate its arguments. In that case, one approach, which we used extensively in [Friedman & Wand 84], was to wrap the values in quote. This fails in the current context because we would like to define quote using make-reifier. An approach which does work is to pass to bf three identifiers and an environment in which those identifiers are bound to the right values. This approach is preferable even where quote would work (as in the reflection functions below), because it is no longer dependent on the correct definition of quote, and it furthermore avoids the use of handles [Smith 82]. This leads to the following definition:

```
(define make-reifier
  (let ([ERK '(E R K)])
    (curry (bf e r k mk)
      (mk (C bf ERK
             (extend r ERK
               (list e (U->BF r) (K->BF k))))))))
```

Here E, R, and K are the three identifiers which are bound to the right values. This defines make-reifier as a primitive operation in Scheme, of type $BF \rightarrow BF$. It may then be imported into the initial environment by the techniques we shall explain below.

This code assumes that bf is a simple abstraction. It is possible to do a variety of interesting things by writing code in which bf is not a simple abstraction. For example, the names of the formal parameters E, R, and K may be detected by invoking the following function on no arguments:

```
(set! find-make-reifier-formals
  (make-reifier
    (lambda (a b c)
      ((make-reifier
         (make-reifier
           (lambda (x y z) (c x))))))))
```

In [Smith 82, 84], analogous techniques may be used to detect essentially all of the text of the interpreter; thus, as Smith points out, *any* change to the 3-LISP interpreter, no matter how minor (including change of bound variables) results in a different language. By restricting such access in Brown, we get the benefits of a tower model while maintaining the traditional distinction between the defined language and defining language. By this choice, we learn more about the design space for reflective languages.

5.5 Reflection

We next turn to the reflection functions. These take Brown functions and turn them back into objects of type *K*, or *Env*. As with `make-reifier`, the technical problem here is that the brown function `bf` will typically be a call-by-text function which evaluates its arguments (probably created by evaluating an expression of the form `(lambda (...) ...)`). As we did with `make-reifier`, we solve this problem by passing to the function an identifier as an actual parameter, along with an environment in which that identifier is bound to the correct value.

```
(let ([z '(v)])
  (define BF->K
    (curry (bf v)
      (C bf z
         (extend global-env z (list v)) initk)))
  (define BF->U
    (curry (bf v)
      (C bf z
         (extend global-env z (list v))))))
```

These functions are used when we start a lower interpreter. This is done via the function `meaning` which takes a list of Brown values representing an expression, an environment, and a continuation, and which starts a new interpreter. The continuation at the time the new interpreter is started is built into the new metacontinuation, as in `K->BF`:

```
(define meaning
  (curry (erk k mk)
    (C denotation
       (first erk)
       (BF->U (second erk))
       (BF->K (third erk))
       (C meta-cons k mk))))
```

5.6 The tower

We are now ready to write the read-eval-print loop and the tower. We rewrite the tower here in our Curried style.

```
(define R-E-P
  (lambda (prompt)
    (Y (curry (loop v)
         (C denotation
            (prompt&read
              (print&prompt prompt v))
             global-env
             loop)))))

 (define tower
   ((Y (curry (loop n theta)
         (C theta (R-E-P n) (loop (add1 n)))))
    0))
```

We also define a version of `readloop` suitable for importing as a primitive into Brown:

```
(define readloop
  (lambda (prompt)
    (K->BF (R-E-P prompt))))
```

We start the system by calling `boot-tower`:

```
(define boot-tower
  (lambda ()
    (C initk 'starting-up tower)))
```

5.7 The Initial Environment

Before we can start the tower, we must supply it with a suitable global environment, which will be shared by all the interpreter levels.

We first define `extend`, which extends a given environment by binding a list of names to new cells containing a list of values. This is relatively routine; the only coding trick we have performed is to use a function `rib-lookup` which takes a name to be looked up, a list of names, a list of corresponding cells, a success continuation to which the matching cell is to be sent, and a failure continuation (a function of no arguments) to be invoked in case of failure:

```
(define extend
  (lambda (r names vals)
    (if (= (length names) (length vals))
        (let ([cells (map make-cell vals)])
          (curry (name k)
            (rib-lookup name names cells k
              (lambda () (C r name k)))))
        (wrong (list "extend: "
                     "Formals: " names
                     "Actuals: " vals)))))

(define rib-lookup
  (lambda (id names cells sk fk)
    (C Y (curry (lookup names cells)
           (cond
             [(null? names) (fk)]
             [(eq? (first names) id)
              (sk (first cells))]
             [else
              (C lookup
                 (rest names)
                 (rest cells))]))
       names cells)))
```

We choose to import values from Scheme by name. To do this, we use the function `id->BF`. This takes an identifier, finds its global binding in Scheme, converts it to an element of *F* (a function that takes a list of arguments and a continuation), and then converts that to a simple Brown function:

```
(define id->BF
  (let ([host->F
          (curry (f v* k) (k (apply f v*)))])
    (lambda (x)
      (F->BF (host->F (host-value x))))))
```

We can now describe the creation of the initial environment. The function boot-global-env creates an initial rib, consisting of a list of names and a corresponding list of cells containing the appropriate values. The name list consists of a few special cases along with a list of names, called primop-name-table, of functions that are to be imported from the host. Corresponding to these names it creates a list of cells; for the imported functions, the values are imported using id->BF.

The function global-env is then created; it is a function which merely calls rib-lookup with this initial rib and with a failure continuation which specifies what to do in case of a lookup of an identifier which does not appear in the global environment. This failure continuation adds a cell to the global environment corresponding to the previously unknown identifier. This allows us to accomodate run-time extension of the global environment, as in the definition of if above.

```
(define boot-global-env
  (let ([id->F-cell
          (lambda (x) (make-cell (id->BF x)))])
    (lambda ()
      (let ([initnames
              (append
                (list 'nil 't 'wrong 'meaning)
                primop-name-table)]
            [initcells
              (append
                (map make-cell
                     (list 'nil 't
                           (K->BF wrong)
                           (F->BF meaning)))
                (map id->F-cell
                     primop-name-table))])
        (define global-env
          (curry (id k)
            (rib-lookup
              id initnames initcells k
              (lambda ()
                (let
                  ([c (make-cell 'UNASSIGNED)])
                  (set! initnames
                        (cons id initnames))
                  (set! initcells
                        (cons c initcells))
                  (k c))))))))))
```

5.8 Side Effects

A purely functional language which had no side-effects whatever (including printing its results!) would be useless, as it would have no capability for interaction with the real world. Hence it is necessary to build some side-effecting interface. The key problem in managing this interface is the need to make sure that operations with side-effects are done at the right time. In an applicative-order language like Scheme, this is done by wrapping each possibly destructive operation in a lambda; we are then assured that the operation is not performed until the function is applied. In our case, we wrap each destructive operation in (lambda (mk) ...), so no side-effect is performed until the denotation is really applied to a metacontinuation. Thus we report errors using

```
(define wrong
  (curry (v mk)
    (writeln "wrong: " v)
    (C initk 'wrong mk)))
```

and the error will not be reported until (wrong v) is applied to a metacontinuation. Similarly, since arbitrary functions imported from Scheme may have side-effects, we made sure to write

```
(curry (v* mk) (C fun v* k mk))
```

in the definition of F->BF; since F->BF is used as part of the importation process, this assures that no imported primitive is executed prematurely.

6. Are Metacontinuations Necessary?

One might ask whether the introduction of metacontinuations is necessary, as they are not reifiable and the tower maintains strict stack discipline: there is nothing in the tower like the non-local jumps that mandated the introduction of conventional continuations in the interpreter. One can, in fact, formulate a plausible "direct" semantics for towers. In this semantics, rather than having $\mathcal{E}$ be tail-recursive with the metacontinuation appearing as an argument, we would keep the old functionality of $\mathcal{E}$ and have the lower interpreter spawned non-tail-recursively, via something like

$$\mu(\mathcal{E}[\![e]\!]\rho\kappa)$$

The initial metacontinuation (the tower) would be constructed as the value of

$$\mu_\infty = Y(\lambda\mu.\mu(\mathcal{E}[\![e_0]\!]\rho_0\kappa_0))$$

This semantics would be far more appealing in Smith's methodology, as it would avoid introducing a non-reifiable component. Unfortunately, the term for μ_∞ is an unsolvable term of the λ-calculus: it has no head normal form. Thus it denotes the bottom element in any sensible model of the λ-calculus [Barendregt 81], including all the standard models. Hence, making this semantics non-trivial would require a very non-standard model of the λ-calculus.

7. Conclusion

We have given a semantic account of Smith's tower of metacircular interpreters by introducing a new context component, called a metacontinuation, that abstracts the state of the tower above the current interpreter. This account is purely functional, makes a minimum of implementation decisions, and does not employ reflection to explain reflection.

Acknowledgements

The authors wish to acknowledge the contributions of Bruce Duba to this paper. In particular, the self-defining `set!` and the trick for avoiding the use of quote are his. Thanks also to Matthias Felleisen for his comments.

References

[Barendregt 81]
Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, Amsterdam, 1981.

[Felleisen *et al.* 86]
Felleisen, M., Friedman, D.P., Kohlbecker, E., and Duba, B. "Reasoning with Continuations" *Proc. First Ann. IEEE Symp. on Logic in Computer Science* (Cambridge, MA, June, 1986).

[Friedman & Wand 84]
Friedman, D.P., and Wand, M. "Reification: Reflection without Metaphysics" *Proc. 1984 ACM Symposium on Lisp and Functional Programming* (August, 1984), 348–355.

[Kohlbecker 86]
Kohlbecker, E. *Syntactic Extensions in a Lexically Scoped Language*, Ph.D. dissertation, Indiana University, to appear.

[Smith 82]
Smith, B.C., *Reflection and Semantics in a Procedural Language*, MIT-LCS-TR-272, Mass. Inst. of Tech., Cambridge, MA, January, 1982.

[Smith 84]
Smith, B.C., "Reflection and Semantics in Lisp," *Conf. Rec. 11th ACM Symp. on Principles of Programming Languages* (1984), 23–35.

[Smith & des Rivières 84]
Smith, B.C., and des Rivières, J. "The Implementation of Procedurally Reflective Languages," *Proc. 1984 ACM Symposium on Lisp and Functional Programming* (August, 1984), 331–347.

Appendix: Help Functions

This appendix lists all the help functions necessary to make the code in the text runnable.

```
; applicative-order Y combinator

(define Y
  (lambda (f)
    (let ([d (lambda (x)
               (f (lambda (arg)
                    (C x x arg))))])
      (d d))))

; decomposing expressions

(define first car)
(define second cadr)
(define third caddr)
(define rest cdr)


; cells

(define deref-cell car)
(define make-cell (lambda (x) (cons x '())))
(define set-cell!
  (lambda (x y) (set-car! x y) y))

; input/output with prompts

(define prompt&read
  (lambda (prompt)
    (print prompt) (print "-> ") (read)))
(define print&prompt
  (lambda (prompt v)
    (writeln prompt ":: " v) prompt))


; find the global binding of an identifier

(define host-value
  (lambda (id) (eval id)))

; list of names to import from host

(define primop-name-table
  (list 'car 'cdr 'cons 'eq? 'atom? 'symbol?
        'null? 'not 'add1 'sub1 'zero? '+ '- '*
        'set-car! 'set-cdr!
        'print 'length 'read 'newline 'reset
        'make-cell 'deref-cell 'set-cell!
        'ef 'readloop 'make-reifier))
```