Lisp and Symbolic Computation, 1, 11-37 (1988)

The Mystery of the Tower Revealed: A Nonreflective Description of the Reflective Tower

MITCHELL WAND
College of Computer Science, Northeastern University, Boston, MA 02115

DANIEL P. FRIEDMAN
Computer Science Department, Indiana University, Bloomington, IN 47405

Abstract

In an important series of papers [8, 9], Brian Smith has discussed the nature of programs that know about their text and 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 metacircular 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 metacircular 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 [4], we showed how a useful species of reflection could be modeled without the use of towers. In this paper, we give a semantic account of the reflective tower. This account is self-contained in the sense that it does not employ reflection to explain reflection.

1. Modeling reflection

Reflective programming languages were introduced in [8, 9] to study programs that need knowledge of their own behavior. In artificial intelligence, this kind of knowledge is needed, for example, in programs that must explain their behavior to a user. In the study of programming languages, a similar phenomenon occurs in extensible languages, that is, languages in which one can write programs that change the language itself. Lisp is such a language: The user can change the language itself by defining new special forms, which effectively cause new lines to be added to the Lisp interpreter.

In [4], we constructed a formal model of this behavior. To understand this model, 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 that includes several parts:

A preliminary version of this paper appeared in *Conference Record of the 1986 ACM Conference on Lisp and Functional Programming,* Cambridge, MA, August 1986. This material is based on work supported by the National Science Foundation under Grants MCS 8303325, MCS 8304567, and DCR 8605218.

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 modeled 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

$$: Exp\text{-- } Env \text{ —}K \text{ -. } A \; = \; \text{Xepx. .}$$

where A is some domain of answers. Thus we can think of (once we have written it out) as defining an interpreter that manipulates three registers, e, p, and K.

Now we can define new language constructs. Consider adding a new kind of expression, called an **add-immediate** expression, which has two subexpressions. The first is an arbitrary expression, which is to be evaluated, and the second is a number. The result of the entire expression is to be the sum of the number and the value of the subexpression. We can define this in the language of denotational semantics as follows:

$$\textbf{(add-immediate } e, n_z)\sim \; = \; \text{Xpx.f;' F[e,]Ip(Xu.x(n}_2 + \text{u))}$$

In the system described in this paper, we might define it by

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

The intention is that when an `add-immediate` expression (say `(add-immediate x 3))` is evaluated, `e will` be bound to the arguments of the expression (in this case the list `(x 3)), r will` be bound to the current environment, and k will be bound to the current continuation. Then the body of the definition (the `meaning` expression) will be executed. We refer to this process as *reification.* When reification occurs, the contents of the interpreter registers, e, p, and x, 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. We refer to a procedure of this sort as *a reifying procedure.*

Conversely, *reflection* is the process by which values for an expression, an environment, and a continuation are reinstalled as the values of the interpreter registers. This process may be thought of as turning data into program. In our example, the function `meaning` takes its arguments and installs them in the interpreter registers, so that the first subexpression (in the case of `(add- immediate x 3)`, the identifier x), is loaded into the expression register, the old environment is reloaded into the environment register, and a suitable continuation is loaded into the continuation register.

These transformations may be summarized as follows. Let us assume that the identifier f is bound to a reifying procedure that might be represented as a closure consisting of a body e_0, a static environment ρ_0, and formal parameters `(e r k)`. Then reification may be summarized as

$$[\![(f\ e_1\ e_2 \ldots e_n)]\!] \rho\kappa = [\![e_0]\!] (\rho_0 [e \mapsto (e_1\ e_2 \ldots e_n),\ r \mapsto \rho^{\wedge},\ k \mapsto \kappa^{\wedge}]) \kappa_0$$

where $\rho^{\wedge}$ and $\kappa^{\wedge}$ are suitably reified versions of ρ and κ, and κ_0 is an arbitrary continuation.

Reflection, on the other hand, can be summarized as follows:

$$\begin{aligned} &[\![(\texttt{meaning}\ e_1\ e_2\ e_3)]\!] \rho\kappa \\ &= meaning\ (e,\ \rho_r,\ \kappa_r)\ \kappa \\ &= (\lambda\ e\rho;\ ^{\vee}\kappa^{\vee}_{t} \end{aligned}$$

Here the procedure `meaning` is called on actual parameters e_1, e_2, and e_3. These parameters are evaluated, yielding values e, ρ_r, and κ_r, which are an expression, a representation of an environment, and a representation of a continuation, respectively. These values are transformed back into real environments and continuations by the "down" operator $(\cdot)^{\vee}$, installed in the interpreter registers, and the computation proceeds.

In this model, reifying procedures are ordinary values, like regular procedures. Thus, this model generalizes 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.

On the other hand, this model displays some disturbing asymmetries. If a reification is followed by a reflection, then the choice of the continuation κ_0 is arbitrary; one doesn't care what happens when the body of a reifying procedure "runs off the end" without reinvoking `meaning`. Furthermore, if a reflection is followed by a reification, one loses the context κ in which the original reflection occurred, and the choice of κ_0 may become significant.

These pathologies led us to consider the question of providing a reasonable model of the tower itself. Reflective models (as in [9]) 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 nonreflective models ([2, 8: Chapter 5]) were extremely operational. Indeed, it was not clear whether the techniques of denotational semantics were adequate to describe the tower.

Though Smith has argued eloquently for the desirability of reflective languages, the lack of a denotational characterization has been an impediment to a deeper understanding of the implications of reflection. In this paper, we attempt to remedy this difficulty by showing how denotational semantics can be used to describe a tower of computations. Such a model may help provide a setting for evaluation of the reflection concept.

2. **Modeling the tower**

3-LISP adds to the reflective structure a serious commitment to the idea of metacircular 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 that spawned it. When the lower interpreter invokes a reifying procedure, its registers are reified and passed to the body of the procedure, as in our earlier model. In 3-LISP, however, the body of the reifying procedure is then executed *as if it were in the upper interpreter.* Thus, reifying procedures in 3-LISP, like Lisp s special forms, effectively add new lines to the interpreter.

Our standard treatment of contextual information gives a straightforward way of modeling 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 (is now

: *Exp—.Env—K—). MK —A*

A metacontinuation represents the state of the upper interpreter, and by implication that of the tower above it. Thus execution always takes place in the bottom interpreter of the tower.

We may now look at the transformations for reification and reflection in the presence of a metacontinuation. Reification looks like

[[(f *e, ...* e„)]l pK N = Deo]I(po[e *F (e, ... en), r F* p^, **k** *F* KA 1) K |

where K, and μ, are a continuation and metacontinuation extracted (in a manner yet to be determined) from Reflection becomes

f,' If(meaning *e, ez e*3)]]pKM

= *meaning (e,* p,, K,) κμ

~` ~e ~ Pi:x 4t,

where p, is a new metacontinuation encapsulating x and p. If the extraction and encapsulation processes are made invertible, then reification and reflection will become symmetrical: reification passes representations of both the environment and continuation to the upper interpreter, which can then dynamically recreate the lower interpreter by performing the appropriate reflection.

The first problem is characterizing the domain MK. We may think of a metacontinuation as waiting for a result from the lower interpreter. Thus, MK will have the form

$$MK=R\text{-}3A$$

where R is the domain of interpreter results.

Our next task is to determine the domain R. To do this, we will consider what happens when the lower interpreter invokes a reifying procedure and returns to the next level.

When the lower interpreter executes a reifying 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 we see that combining an expression and an environment gives us an object of type K -~ MK A. When a reflective procedure is invoked, the appropriate thunk is built and passed to the metacontinuation. Thus the domain of metacontinuations should be

$$MK= [K—MK\text{-}\sim A]\text{-}\sim A$$

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, p, and K, 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 x and metacontinuation p, we expect it to return a thunk 6 which will be run on continuation x and metacontinuation p. Thus, the lower interpreter should be run with metacontinuation

XO.OKp

That is, the operation of building this new metacontinuation is

meta-cons = ~,Kp6.OKp

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 [l, 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:

$$g\text{-} = Y(X\ .i.meta\text{-}cons\ \mathrm{K_o}\ \mu)$$

where Ko 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 that is a read-eval-print loop.

The only two legal operations on metacontinuations are the operations of pushing and popping continuations off the metacontinuation. To ensure that these are the only operations we perform, we introduce two new abstractions for performing these operations.

The first is shifting up, or sending a thunk to the current metacontinuation. If the metacontinuation was built by `meta-cons`, then the thunk will be run with the continuation and subsequent metacontinuation (the "meta-car" and "meta-cdr") of the metacontinuation. We express this as follows:

```
(define shift-up
  (lambda (theta)
    (lambda (mk)
      (mk theta))))
```

The argument `theta` of `shift-up` is a function that expects a continuation with which to run some code. It then takes a metacontinuation. The metacontinuation contains the continuation from "the level above." Access to this continuation is obtained by invoking the metacontinuation on `theta`. The type of `shift-up` is thus *[K~MK—A] -->MK –. A.*

By expanding the definition of `meta-cons`, we may deduce

```
((shift-up theta) ((meta-cons k) mk)) = ((theta k) mk)
```

We can use `shift-up` to define extraction operations as well:

```
(define meta-car (shift-up (lambda (v) (lambda (mk) v))))

(define meta-cdr (shift-up (lambda (v) (lambda (mk) mk))))
```

By expanding these definitions, we may deduce

```
(meta-car ((meta-cons k) mk)) = k
(meta-cdr ((meta-cons k) mk)) = mk
```

Conversely, we could have defined `meta-car` and `meta-cdr` by explicitly passing the appropriate functions to `mk`:

```
(define meta-car
  (lambda (mk)
    (mk (lambda (k) (lambda (mk) k)))))

(define meta-cdr
  (lambda (mk)
    (mk (lambda (k) (lambda (mk) mk)))))
```

Then we could define `shift-up` in terms of `meta-car` and `meta-cdr`:

```
(define shift-up
  (lambda (th)
    (lambda (mk)
      ((th (meta-car mk)) (meta-cdr mk)))))
```

These relationships are useful in understanding the use of `shift-up` in the code of the model later.

The other shifting operation is shifting down, that is, starting a new lower level by pushing the current continuation onto the metacontinuation and running in a different continuation. We express this as follows:

```
(define shift-down
  (lambda (d)
    (lambda (k)
```

```
      (lambda (mk)
        (d ((meta-cons k) mk))))))
```

The first argument to `shift-down` is the code to be run at "the level below." To shift down we must have access to the current continuation that will be pushed onto the tower. Hence, the next argument to `shift-down` is a continuation. We can think of `((shift-down d) k)` as an operation that pushes the continuation `k` onto the tower and starts a new interpreter specified by `d`.

The relationships between `shift-up` and `shift-down` may be expressed by the following equations, which can be derived easily from the definitions:

```
((shift-up (shift-down d)) ((meta-cons k) mk)) = (d ((meta-cons k) mk))

(shift-down (shift-up theta)) = theta
```

Thus, in the context of a metacontinuation formed by `meta-cons`, these two functions are inverses.

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 x waiting in the upper interpreter. Thus we must pass to the metacontinuation a thunk that given x, passes v to it. This can be done by invoking a continuation `terminate-level` defined as follows:

```
(define terminate-level
  (lambda (v)
    (shift-up (lambda (k) (k v)))))
```

3. 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 14]. 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 three-argument abstraction and turns it into a reifying procedure that, when called, reifies the registers e, p, and x into Brown values, creates a suitable thunk, and passes it to the metacontinuation. Such a reifying 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 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 that evaluates either `exp1` or `exp2`, depending on the value returned by `exp0`. The code uses the "extensional if' function `ef` [9], which takes a Boolean and two values and returns one of the two values. Unlike `if`, `ef` is strict in all its arguments; 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 that 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 that takes a prompt and produces a Brown continuation (see Section 4.6 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-with-current-continuation [7]`, which we abbreviate `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 [des Rivieres, 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 [3]?

```
(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 reinvoking 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 that, 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.

4. 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.

4.1. *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 that allow us to proceed without fully parenthesizing all the applications and nested lambdas. We do this using the macro-declaration tool `declare-syntax` [5, 6].

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

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

With these, we can rewrite `meta-cons` as

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

4.2 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

$$Exp \qquad Env\text{–}0K\text{—}MK\text{—}A$$

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 modeling 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)))))))
```

To accomodate reification, Brown uses call-by-text. A Brown function has functionality

$$BF=Exp^{*}\text{->}Env\text{—}K\text{->}MK\text{—}A$$

It gets the text of the actual parameters, the call-time environment, and the call-time continuation and metacontinuation, 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 that 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-~*** K ***-3*** MK *A)* and turns it into a Brown function that 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).

4.3. *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 that can be manipulated [4]. An environment is turned into a one-argument Brown function that evaluates its argument and passes the result (the evaluated actual parameter) to the environment:

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

Continuations are treated similarly. Here `kl` is the continuation to be converted, and `e`, `r`, and `k` are the Brown interpreter's registers at the point that `kl` is invoked. Since a continuation is regarded as restarting a lower interpreter, we save the continuation `k` by putting it in the metacontinuation with `shift-down`, as discussed in Section 2:

```
(define K->BF
  (curry (kl e r k)
    (if (= (length e) 1)
        (C denotation (first e) r
           (lambda (v) (C shift-down (ki v) k)))
        (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 [4], we simply threw away the continuation `k` corresponding to the point that `(K->BF kl)` was invoked.

4.4. *Building reaming procedures*

We need to write a function that takes a simple Brown function and converts it into a reifying 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)
    (shift-up (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 [4], was to wrap the values in quote. This fails in the current context because we would like to define quote using make- reif ier. 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 [8]. This leads to the following definition:

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

Here E, R, and K are the three identifiers that are bound to the right values. This defines make-reifier as a primitive operation in Scheme, of type *BF*—~ *BF*. It may then be imported into the initial environment by the techniques explained in Section 4.7.

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 expression:

```
((make-reifier
  (lambda (a b c)
    ((make-reifier
      (make-reifier
        (lambda (x y z) (c x))))))))
```

In [8, 91, 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.

4.5. 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 that evaluates its arguments (probably created by evaluating an expression of the form (lambda (. . .) ...)). As we did with make - reifier,wesolvethisproblembypassing to the function an identifier as an actual parameter, along with an environment in which that identifier is bound to the correct value:

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

Recall that a continuation takes a value and a metacontinuation as its arguments. Thus, (BF->K bf) should be a continuation. It takes a value and then invokes shift-up. shift-up, in turn, takes the current metacontinuation as an argument, and runs a thunk which invokes the Brown function bf on the appropriate environment, using the continuation at the top of the metacontinuation (i.e., at the bottom of the tower). A slightly less roundabout version of the code might be

```
(define BF->K
  (let ([z '(v)])
    (curry (bf v mk)
      (C bf z (extend global-env z (list v))
         (meta-car mk)
         (meta-cdr mk)))))
```

where meta-car and meta-cdr extract the appropriate pieces from the metacontinuation.

We write BF->U similarly; it is less complicated because no shifting is necessary.

```
(define BF->U
  (let ([z '(v)])
    (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)
    (shift-down
     (C denotation
        (first erk)
        (BF->U (second erk))
        (BF->K (third erk))))))
```

4.6. *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 (add 1 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 terminate-level 'starting-up tower)))
```

47. 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. In this code, a call of `extend` with unequal numbers of names and values, produces an environment which signals an error whenever it is invoked. This error could be detected at the time the environment is created by writing `extend` itself in continuation-passing style, and modifying all the invocations of `extend` appropriately.

```
(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)))))
        (curry (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 that 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 that 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 terminate-level)
                           (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)))))))))
```

4.8. *Side effects*

Our language communicates with the outside world through side effects (such as `set!` and the read-eval-print loop). The key problem in managing side effects 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 terminate-level '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.

5. Are metacontinuations necessary?

One might ask whether the introduction of metacontinuations is neccessary, since they are not reifiable and the tower maintains strict stack discipline: there is nothing in the tower like the nonlocal 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 f be tail-recursive with the metacontinuation appearing as an argument, we would keep the old functionality of (and have the lower interpreter spawned non-tail-recursively, via something like

$$g(\sim` QebK)$$

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

$$= Y(a.g.\mu \quad E(e_{ollPo}{}^{K}{}_{0}))$$

This semantics would be far more appealing in Smith's methodology, as it would avoid introducing a nonreifiable component. Unfortunately, the term for g~ is an unsolvable term of the X-calculus. Thus it denotes the bottom element in any model of the X-calculus that is *sensible [1].* The class of sensible models includes all the standard models. Hence, making this semantics nontrivial would require a very nonstandard model of the A.-calculus.

6. Is lambda necessary?

In Section 3 we showed how the reifier `set !` could be defined in Brown, using suitable primitive store manipulations and the reification mechanism. In this section we do the same for `lambda`: we show how `lambda` can be *defined* in Brown. Thus we could have presented Brown in Section 4 using only identifiers, application, and some reifying primitives; without, in particular, having a line in `denotation` for abstractions.

We do this in two steps. First, we show how to write a reifier that builds a one-argument call-by-value abstraction, just like the one built by `lambda`. Then we show how to eliminate `lambda` from that definition, so that it is properly bootstrappable.

Since one can write a denotational definition for a `lambda` term, one can transcribe that definition into the definition of a reifier, just as we did for `set!`. If we do that, we get the following:

```
(set! lambda-value
  (make-reifier
    (lambda (e r k)
      (k (make-reifier
           (lambda (el rl kl)
             (meaning (car el) rl
               (lambda (val)
                 (meaning
                   (car (cdr e))      ; the body
                   ((lambda (cell-val)
                      (lambda (id)
                        (if (eq? id (car (car e))) ; the formal parameter
                            cell-val
                            (r id))))
                    (make-cell val))
                   kl)))))))))
```

Why is this code not bootstrappable? There are only three problems:

1. We have an occurrence of `if`, which is a defined reifier, not a primitive. This can be replaced by the primitive function `ef` by replacing the `(if . . . )` expression by

```
((ef (eq? id (car (care)))
     (lambda () cell-val)
     (lambda () (r id))))
```

   This idiom is necessary because evaluation of `(r id)` might fail if it is called prematurely.
2. We have some occurrences of three-argument abstractions, all as arguments to `make-reifier`. We can eliminate these by redesigning `make-reifier` to take its argument in curried form.
3. The first two items have studiously ignored the obvious problem: all the occurrences of `lambda` itself. But these may be replaced by suitable combinators, using the well-known techniques of bracket abstraction [1, p. 148]. The result is

a term using only functional combination of the combinators S and K and the free variables `meaning`, `ef`, `eq?`, `make-cell`, `car`, `cdr`. Let us refer to this rather large term as "`--code for creating lambda--`". (Though it is theoretically possible to create this term using only S and K, the term is quite large (several hundred corn-binators), and therefore it was necessary to use an efficient bracket-abstraction algorithm, such as that in [10], when we performed the experiment.)

We can now use `(make-reifier --code for creating lambda--)` anywhere we need something that behaves like `lambda`. For example, we can locally bind `frotz` to act like `lambda` by writing

```
(((make-reifier --code for creating lambda--) (frotz) --body--)
 (make-reifier --code for creating lambda--))
```

just as we did for `set!` in Section 3.

To bootstrap the system, we use the expression for creating `set!`, as we did before, but we first bind `lambda` before executing it:

```
(((make-reifier --code for creating lambda--) (lambda)
  --code for creating set!--)
 (make-reifier --code for creating lambda--))
```

We can then set the value of the identifier `lambda`:

```
(set! lambda (make-reifier --code for creating lambda--))
```

in the usual way. Note that in this system, `lambda` is also a first-class citizen, whereas it was a special form (the only special form!) in the preceding versions. Hence we can now write

```
((lambda (frotz) --body--)
 lambda)
```

which was previously illegal. So in the presence of reification (and suitable functional primitives), it is not necessary to have *any* built-in special forms: they can all be defined.

Of course, other versions of abstraction can be defined in the usual way using reifiers. Call-by-name, for example, could be defined as follows:

```
(set! lambda-name
  (make-reifier
    (lambda (e r k)
      ((lambda (var body)
         (k (make-reifier
              (lambda (el rl kl)
```

```
            ((lambda (r2)
             (meaning body r2 k1))
               ((lambda (arg)
                  (lambda (id)
                    (if (eq? id var)
                        ((make-reifier
                           (lambda (u v k2)
                             (meaning arg r1
                               (lambda (x)
                                 (k2 (make-cell x)))))))
                         (r id))))
                (car e1))))))
   (car (car e))
   (car (cdr e))))))
```

7. Conclusions and open problems

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 relies entirely on the nonimperative features of Scheme (except for the real-world interface discussed in Sections 4.7 and 4.8), makes a minimum of implementation decisions, and does not employ reflection to explain reflection. A number of open problems remain, however.

The presence of the tower gives yet another dimension to our design decisions. As discussed in [4], one has considerable latitude in choosing how interpreter objects are to be reified. With the tower, one has the additional choice of what level of the tower should be used for the invocation of these objects. Some of our code incorporates arbitrary choices on this issue. The rationale for these choices needs to be understood more deeply.

A related issue is the algebra of reflection. The abstractions `shift-up` and `shift-down` obey a number of algebraic identities, some of which we discussed above. These identities, along with the related identities for A and ", seem to be helpful in understanding the behavior of reflective systems. We need to understand which of the potential identities are important, and to develop reification strategies that fully exploit this algebraic behavior.

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 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))
```

Loading Files (note added in proof)

One shortcoming of the system as presented here is that it is unable to perform a load-file operation. The obvious solution, that of starting a new interpreter with the read operation appropriately modified, does not work, because expressions read from the file may cause subsequent expressions to be read by higher levels of the tower. Thus reading from a file requires more cooperation between interpreter levels than is easily accomplished in our model.

While such cooperation can in principle be obtained by suitable modification to the system, the following interim solution, due to Kevin Likes and Julia Lawall, will enhance the usability of the system. First, redefine `prompt&read` and `R-E-P` as follows:

```
(define prompt&read
  (curry (prompt k)
    (display prompt) (display "-*") (k (read))))

(define R-E-P
  (lambda (prompt)
    (Y (curry (loop v)
        ((prompt&read (print&prompt prompt v))
         (lambda (v)
           (if (eof-object? v)
               (lambda (mk) '---done---)
               (C denotation v global-env loop))))))))
```

Then add the following:

```
(define brown-load
  (lambda (file)
    (with-input-from-file file boot-tower)))
```

and add `brown-load` to `primop-name-table`. When `brown-load` is invoked, it causes Scheme to start an entirely new tower with its input at all levels taken from `file`. The new tower communicates with the old one by side-effecting the shared store. When an end-of-file is read, the new tower terminates and returns `---done---` as its answer, which is reported to the old tower as the value of the call.

Acknowledgments

The authors 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. Jim des Rivieres patiently explained many of the difficult corners of 3-LISP to us, and provided extensive comments on [4] and several versions of this paper. Thanks also to Matthias Felleisen for his comments.

References

1. Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amersterdam, 1981.
2. des Rivieres, J., and Smith, B.C. The Implementation of procedurally reflective languages. Conf. Rec. 1984 ACM Symp. on Lisp and Functional Programming, Austin, Texas, August 1984, pp. 331-347.
3. Felleisen, M., Friedman, D.P., Kohlbecker, E., and Duha, B. A syntactic theory of sequential control. *Theoretical Computer Science,* ***52*** (1987), 205-237. Preliminary version appeared as: Reasoning with continuations. *Proceedings, First Annual IEEE Symposium on Logic in Computer Science,* Cambridge, MA, June 1986, pp. 131-141.
4. Friedman, D.P., and Wand, M. Reification: Reflection without metaphysics. Conf. Rec. 1984 ACM Symp. on Lisp and Functional Programming, Austin, Texas, August 1984, pp. 348-355.
5. Kohlbecker, E. *Syntactic Extensions in a Lexically Scoped Language.* Ph.D. dissertation, Indiana University, Bloomington, 1986.
6. Kohlbecker, E., and Wand, M. Macro-by-example: Deriving syntactic transformations from their specifications. Conf. Rec. 14th ACM Symp. on Principles of Programming Languages, Munich, Germany, January 1987, pp. 77-84.
7. Rees, J., and Clinger, W., Eds. Revised^3 report on the algorithmic language Scheme, *SIGPLAN Notices* **21,** 12 (Dec. 1986), 37-79.
8. Smith, B.C., *Reflection and Semantics in a Procedural Language.* MIT-LCS-TR-272, MIT, Cambridge, MA, T982.
9. Smith, B.C., Reflection and semantics in Lisp. Conf. Rec. 1 lth ACM Symp. on Principles of Programming Languages, Salt Lake City, Utah, January 1984, pp. 23-35.
10. Turner, D.A. A new implementation technique for applicative languages. *Software-Practice and Experience* ***9*** (1979), 31-49.