MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo 946

June 1988

Reification without Evaluation

Alan Bawden

Abstract

Constructing self-referential systems, such as Brian Smith's 3-Lisp language, is actually more straightforward than you think. Anyone can build an infinite tower of processors (where each processor implements the processor at the next level below) by employing some common sense and one simple trick. In particular, it is *not* necessary to re-design quotation, take a stand on the relative merits of evaluation vs. normalization, or treat continuations as meta-level objects. This paper presents a simple programming language interpreter that illustrates how this can be done. By keeping its expression evaluator entirely separate from the mechanisms that implement its infinite tower, this interpreter avoids many troublesome aspects of previous self-referential programming languages. Given these basically straightforward techniques, processor towers might be easily constructed for a wide variety of systems to enable them to manipulate and reason about themselves.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

This is a revision of a paper that will appear in the *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming.*

©1988 Association for Computing Machinery.

1 Introduction

In [4], and [5], Smith presents the 3-Lisp language as an example of an architecture that supports self-referential properties in a programming language. 3-Lisp is a significant departure from traditional Lisp dialects. Even before self-referential mechanisms are introduced, Smith engages in a reconstruction of Lisp's notions of evaluation and quotation. Then self-reference is introduced through the mechanism of a "tower" of processors—each processor runs an interpreter to implement the processor at the next level below. Each step in the development of 3-Lisp raises interesting questions about programming language design and implementation.

In [2] and [7] Wand and Friedman set an excellent example by attempting to dissect the 3-Lisp language into parts. In [2] the notion of "reification", the mechanism by which the state of a process is made available to the process itself, is separated from the rest of the mechanisms of 3-Lisp. In [7] they show how to add a tower of interpreters to their model. All of this is done without a reconstruction of evaluation and quotation.

The construction of towers of processors seems like it might be a promising technique for constructing systems that have flexible access to their own implementation, but two aspects of such towers as constructed by Smith and Wand and Friedman are troublesome: First, they rely on expressions and environments as the basic language for describing processes. Second, continuations are treated as second-class objects and are overloaded with an additional role as agents for shifting between levels of the tower.

In this paper I will examine these problems in detail, relying on Wand and Friedman's interpreters to illustrate the difficulties. Then I will show how these problems can be addressed by exhibiting an infinite tower of processors in which expressions are not explicitly manipulated, and in which continuations are not used for level-shifting. This will demonstrate that reification and the notion of a tower of processors are independent of the notion of a Lisp expression interpreter.

The resulting language, written in Scheme [3], is called "Stepper".

2 Expressions and environments

In [2], Wand and Friedman describe their language "Brown", as a demonstration of how the data structures of an interpreter can be made available to the program it is interpreting. In essence this is an implementation of "fexprs" with two additional features: First, Brown's fexprs receive a continuation as an explicit argument, in addition to receiving an expression and an environment. Second, Brown's fexprs can be passed as arguments to other procedures as if they were ordinary procedures.

In addition, the representation of an environment used by Brown is procedural. Thus it is possible to call the Brown evaluator with an environment that performs some arbitrary computation when a variable is accessed.

Brown is an important step in dissecting 3-Lisp because it separates reification from the notion of a tower of interpreters, and because it supports reification without resorting to a 2-Lisp-style reorganization of Lisp. This is especially important because Brown almost precisely separates out the objectionable features of 3-Lisp: the introduction of a fexpr mechanism, and the second-class treatment of continuations.

By examining the Brown theory of reification, which seems to be a faithful model of 3-Lisp's reification, we can hope to learn how to replace that part of 3-Lisp with something less objectionable. We might learn how to construct something like the 3-Lisp tower of processors in which a reifying procedure sees some data structures that are more well-behaved than expressions and environments.

We will deal with the issue of continuations in more detail in the next section, since Brown's treatment of continuations changed in [7] when a tower of interpreters was introduced. For the moment it suffices to observe that continuations do not play a very important role in this original version of Brown. They serve as a second example, after environments, where Wand and Friedman have made design choices about the representation of reified objects, but continuations do not need to be be reified at the same time environments are, nor is it even necessary for the implementation of Brown to spend as much time as it does worrying about managing them.

To see this more clearly, observe that the interpreter's own procedures pass explicit continuations among themselves solely so that they can easily be passed on to reifiers. The interpreter procedures are always careful to call each other tail-recursively; the *same* implicit continuation is always passed from one interpreter procedure to another. Since the implicit continuations aren't being used for anything, it is an easy exercise to re-write this version of Brown so that it relies on implicit continuations, and uses the ordinary Scheme calling mechanism to return values.

This changes the protocol for reifiers so that instead of invoking an explicitly passed continuation in order to return a value, they must return the value in the ordinary way (to their implicitly passed continuation). Almost all the reifiers in [2] can be easily rewritten to use this new protocol. The only exception is `call/cc`, which must now be provided as a primitive, implemented using the host language version of `call/cc`. Having provided `call/cc`, any other reifiers that need explicit access to continuations need only call it.

This change would avoid several embarrassing issues, such as what should happen when a reifier returns a value instead of invoking its continuation. These issues are acknowledged in [2] as alternative design choices in the representation of continuations. To re-write Brown in this manner would shift these issues to being design choices in `call/cc`, which is where we are used to encountering them.

Separating the mechanism for reifying continuations from the mechanism for reifying expressions and environments would be an improvement to Brown as a programming language, although it would no longer be quite as faithful a reproduction of 3-Lisp's theory of reification. It only makes sense to lump continuations together with expressions and environments, as Brown and 3-Lisp do, if you believe that continuations rightfully *belong* to the interpreter. If, on the other hand, you believe continuations should be first-class objects that can be safely manipulated by ordinary programmers (unlike expressions and environments, which can easily be misused), and which belong to the running program just as much as the ordinary arguments to a procedure do, then it seems most inappropriate that the same primitive mechanism is used to obtain access to both continuations and expressions and environments.

We now turn to the representations used in Brown for expressions and environments. Expressions are represented in the traditional manner as Lisp lists and symbols. Environments are represented as a procedure that takes an identifier and returns an L-value. An L-value is simply an ordinary `cons`-cell whose `car` contains the value. This is a departure from 3-Lisp where environments are represented using an ordinary A-list.

Brown's procedural representation of environments results in a certain amount of complexity, due to the different calling conventions used by Scheme and Brown. A benefit of this representation is the ability to run some arbitrary code whenever a variable is referenced. This does give reifying procedures some additional leverage in controlling the actions of the evaluator.

Greater leverage could be obtained by representing L-values procedurally as well. This would enable reading a variable to be distinguished from writing a variable. Even more interesting effects might be obtained by representing expressions procedurally. However, it is not clear just what such exercises in procedural representation have to do with the study of reifying interpreters.

If an interpreter uses simple A-lists for environments, as 3-Lisp does, then reifying an environment becomes quite straightforward—no conversion is required at all. Other representations will have other capabilities, and will require more complex conversions. It is an interesting observation that more flexible representations will allow the user correspondingly greater control over the interpreter. However, given that 3-Lisp exhibits the phenomena we are interested in studying *without* resorting to procedural representations, it seems to be an unnecessary distraction for Brown to do so.

A version of the original Brown interpreter which was re-written to eliminate explicit continuations, and in which environments were represented as simple A-lists, would be extremely plain. There would only be two differences between "Simplified Brown" and ordinary Scheme: (1) Simplified Brown would have a mechanism for defining new special forms as reifying procedures, and (2) in Simplified Brown such reifying procedures could be passed as arguments to other procedures. The former is a desirable property for a language to have (if it can be achieved without compromising other aspects of the language), but the latter property is of questionable value. In most other dialects of Lisp it is possible to statically determine which occurrences of an identifier signal special syntax. Eliminating this ability leads to behavior such as that demonstrated in the following dialog with Brown:

```
0-> (set! quote
      (make-reifier
        (lambda (e r k) (k (car e)))))
0:: #<Procedure QUOTE>
0-> (set! call (lambda (f x) (f x)))
0:: #<Procedure CALL>
0-> (call car '(1 2 3))
0:: 1
0-> (call quote '(1 2 3))
0:: X
```

After defining the procedure `quote` as a reifier, we define a procedure `call`, which applies its first argument, a one argument procedure, to its second. `Call` functions as expected when applied to ordinary procedures such as `car`, but when passed a reifier such as `quote` as an argument, its behavior is surprising, and quite unpredictable without examining exactly how `call` was defined. This extreme violation of procedural abstraction is surely the most distressing effect of the kind of reification used by 3-Lisp and Brown. (It is true that this example does not have a direct analog in 3-Lisp due to the way quotation works there, but examples of a similar character can be constructed for 3-Lisp.)

Thus if we leave aside the interesting, but irrelevant, technique of representing environments procedurally, there isn't much about Brown to recommend it as a programming language. It does allow the definition of new special forms in the Brown language itself, but the ability to pass the resulting reifying procedures as arguments is dangerously uncontrolled. Furthermore, continuations, which should be treated as first-class objects, are maltreated by combining the primitive mechanism for obtaining a continuation with the one for obtaining the much more hazardous expressions and environments. Since Brown was designed to behave like 3-Lisp in these aspects, these problems with Brown are problems with 3-Lisp as well.

3 Continuations that shift levels

In [7], Wand and Friedman give a denotational account of a tower of processors. They show how one can construct an infinite tower of interpreter continuations by using the standard fixpoint combinator. An implementation of an infinite tower of interpreters for the Brown language, based on this construction, is presented.

Although [7] is primarily concerned with the denotational semantics of an infinite tower of interpreters, the technique employed by this new Brown interpreter is a very close analog to that employed for 3-Lisp as given in [1]. A Brown meta-continuation is essentially an infinite list of continuations. Meta-continuations are treated as stacks; the only operations performed on them are pushing and popping continuations from the head of the list. This is exactly the way the `state` argument is manipulated by the 3-Lisp implementation.

The installation of an explicit continuation as an implicit continuation by pushing it on to the head of the meta-continuation, or the seizure of a previously implicit continuation by popping it from the meta-continuation, are the level-shifting operations. When a reifier is invoked, the meta-continuation is popped once to obtain the reified, explicit continuation that it requires as an argument. Invoking a reifier thus shifts up a level in the tower. When a reified continuation is called, the continuation implicit at the call is pushed on to the meta-continuation. Invoking an explicit continuation thus shifts down a level in the tower. Again, this is analogous the way that 3-Lisp manipulates its `state` object.

This protocol very neatly deals with all of the embarrassing questions about where the implicit continuation passed to a reifier comes from, and what happens to the implicit continuation when an explicit continuation is invoked. There is a pleasing conservation of continuations. All the design issues surrounding continuations in the original Brown are resolved in constructing the tower.

Unfortunately, although this viewpoint is consistent, it can also be confusing. For example, in [7] two different versions of `call/cc` are defined, and both are incorrect. The first version is defined as follows:

```
0-> (set! call/cc
      (lambda (f)
        ((make-reifier
           (lambda (e r k) (k (f k)))))))
0:: #<Procedure CALL/CC>
```

This version of `call/cc` passes the continuation `k` to the procedure `f`, and also calls `k` on the result, in case `f` returns normally. To demonstrate its behavior we will need to define a procedure `exit` that shifts up the tower once:

```
0-> (set! exit
      (lambda (x)
        ((make-reifier
           (lambda (e r k) x)))))
0:: #<Procedure EXIT>
0-> (exit 'foo)
1:: FOO
```

Having tested `exit`, we are now ready to try `call/cc`:

```
1-> (call/cc (lambda (k) (k '3)))
1:: 3
```

This seems to be correct, as we still seem to be executing at level 1. However, this assumption is false, as we can test by calling `exit` twice:

```
1-> (exit 'bar)
1:: BAR
1-> (exit 'baz)
2:: BAZ
2->
```

In fact, we were running one level deeper in the tower than the prompt led us to believe. The problem is that the continuation `k` had been invoked *twice*, once where `call/cc` did `(k (f k))`, and once when `f` did `(k '3)`. This left two copies of the level 1 loop running, one on top of the other, at the bottom of the tower.

The second version of `call/cc` given in [7] would seem to address precisely this problem:

```
0-> (set! call/cc
      (lambda (f)
        ((make-reifier
           (lambda (e r k) (f k)))))
0:: #<Procedure CALL/CC>
```

Here we avoid calling the continuation `k`, and simply require the user of `call/cc` to always exit by explicitly invoking the continuation he is passed. Unfortunately this restriction is not quite as simple as it sounds. Consider:

```
0-> (call/cc (lambda (k0)
               (call/cc (lambda (k1) (k0 '3)))))
0:: 3
0-> (exit 'foo)
2:: FOO
```

Here `k0` becomes bound to the level 0 loop which is completely removed from the meta-continuation. That is, the procedure passed to `call/cc` is actually run in the interpreter one level up in the tower. Thus `k1` becomes bound to the level 1 loop, which is then lost when `k0` is invoked at level 2. This leaves the level 0 loop running directly below the level 2 loop.

A correct implementation of `call/cc` would not be prone to modifying the tower as these two implementations do. This is not to say that a correct implementation of `call/cc` is impossible in Brown. The 3-Lisp definition of `scheme-catch` found in [5] doesn't have these problems, and it seems likely that the analogous definition would work in Brown as well. Rather, these difficulties are symptomatic of a problem with treating continuations as primitive level shifters.

A much more straightforward view of continuations, treating them on an equal footing with ordinary procedures, would be preferable. Invoking a continuation, like invoking a procedure, should involve no more than a transfer of control within the program running at a single level. Continuations that both transfer control and shift levels make it difficult to reach a particular destination without arriving on the wrong level. Level shifting should not be confused with ordinary transfers of control. If possible, the two kinds of motion should be kept orthogonal.

One way to address this problem would be to imagine that code is converting to continuation-passing style [6] before it is executed. This would assure that continuations and procedures are treated uniformly, since continuations are indistinguishable from procedures after such a conversion. In the next section we shall construct a tower of processors based on this observation. Since continuations will no longer cause diagonal motion through the tower, but will remain confined to a single level, it will be necessary to rely solely on other mechanisms for vertical motion.

4 A simple alternative model: Stepper

Now we turn to the task of demonstrating that the objectionable aspects of 3-Lisp (and the derivative dialects of Brown) discussed above are not inherent in the notion of a tower of interpreters. We shall construct a processor tower in which the levels do not traffic in expressions and environments, and in which continuations are confined to transfer control within a single level.

One possible way to dispose of expressions and environments would be to somehow combine the two into a single object. Reifiers, or whatever passes for reifiers in the new regime, could be passed fully β-substituted structures. A reifier that in Brown would be passed the expression `(car x)` and an environment in which `car` and `x` had the values `#<Procedure CAR>` and `(1 . 2)` would instead be passed (something that behaved like) the expression `('#<Procedure CAR> '(1 . 2))`. This would only be a partial solution to the problem, as the the internal structure of a procedure that invoked a reifier would still be made visible, although not to the extent of revealing the actual names of the identifiers used.

Instead of trying to patch up expressions, Stepper dispenses with expressions entirely. This is a bit of a disappointment, since my original motivation for studying 3-Lisp and Brown was to try and discover something about how fexprs and macros might be tamed. By eliminating expressions from Stepper's theory of reification, we also eliminate the possibility of learning anything new about them.

Expressions are a way of describing a program as constructed out of parts. Programs are made up of `if` expressions, `lambda` expressions, etc. Some processor walks over the program performing some simple actions as it encounters each kind of expression.

Another way to describe a program is as a collection of simple state transitions on some register machine. Each state transition calls for the execution of a few simple operations on the registers, such as moving data from register to register, accessing data in structures addressed by registers, and performing simple arithmetic on the contents of registers. A equally simple processor can push the register machine from state to state performing the operations called for by each state transition.

3-Lisp and Brown view programs as expressions to be evaluated; they break up the execution of a program into a series of calls on `eval`. Stepper views programs as a collection of state transitions; it draws a line through a different part of the interpreter and breaks up program execution into a series of calls on `apply`. Since `apply` is not passed expressions or environments as arguments, Stepper does not have to handle them at all. Furthermore, by treating continuations as just another argument to `apply`, Stepper will avoid giving continuations any special treatment.

At the heart of Stepper is an evaluator with an important difference: wherever an ordinary evaluator would apply a procedure to a continuation and some arguments, or wherever a continuation would be applied to a value, the Stepper evaluator simply makes a list of these items (called a "tuple") and returns them. The toplevel dispatch procedure, `evaluate`, is passed a continuation, an expression, and an environment:

```
(define (evaluate k e r)
  ((cond ((symbol? e) evaluate-identifier)
         ((not (pair? e)) evaluate-constant)
         (else (case (first e)
                 ((quote) evaluate-quote)
                 ((lambda) evaluate-lambda)
                 ((set!) evaluate-set!)
                 ((if) evaluate-if)
                 (else evaluate-application))))
   k e r))
```

Unlike Brown and 3-Lisp, Stepper lacks the property that new special forms can be defined from within the language. Thus all of the special forms of Stepper have dispatch entries in **evaluate**. This is the price we pay for giving up reifiers that manipulate expressions and environments.

Identifiers are evaluated by looking them up in the environment and returning a tuple of the continuation and the value (a complete listing of Stepper, including various utility procedures such as `get-value`, appears as an appendix):

```
(define (evaluate-identifier k e r)
  (list k (get-value e r)))
```

The returned tuple indicates that the *next step* to take in the computation is to apply the first element of the tuple (the continuation) to the rest of the elements of the tuple (the value).

`Lambda`-expressions are evaluated similarly. A tuple is returned that indicates that the next step is to invoke the continuation on an appropriate procedure:

```
(define (evaluate-lambda k e r)
  (define (procedure k1 . vals)
    (evaluate k1
              (third e)
              (extend r (second e) vals)))
  (list k procedure))
```

`Evaluate-lambda` reveals that a Stepper procedure is represented using a Scheme procedure that takes an additional first argument, a continuation. When that Scheme procedure is invoked it will return with the next step in the computation.

Finally, here is the code for evaluating an application:

```
(define (evaluate-application k e r)
  (define (eval-function-continuation f)
    (define (loop args vals)
      (define (eval-arg-continuation val)
        (loop (cdr args) (cons val vals)))
      (if (null? args)
          (list* f k (reverse vals))
          (evaluate eval-arg-continuation
                    (first args)
                    r)))
    (loop (rest e) '()))
  (evaluate eval-function-continuation
            (first e)
            r))
```

This is nothing more that the ordinary function and argument evaluation loop, except that like everything else in the Stepper evaluator, it is careful to return a tuple whenever an ordinary evaluator would call `apply`.

Given one of the tuples returned by `evaluate`, one can advance the computation a single step by applying the first element to the rest of the tuple, obtaining a new tuple in return. A loop like

```
(define (run-loop tuple)
  (run-loop (apply (first tuple) (rest tuple))))
```

is all that is needed drive the computation forward. (`Run-loop` is not actually used by Stepper.)

It is important to realize that the amount of computation performed on each trip through `run-loop` is small and bounded. This is a consequence of the fact that the Stepper evaluator is really converting the source code expression into continuation-passing style on the fly.

Each tuple can be thought of as the current contents of the machine's registers. On each tick the contents of the first register are used to determine how to update the registers. By convention, a procedure is invoked by loading it into the first register, loading a continuation into the second register, and loading its arguments into the following registers. A continuation is invoked by loading it into the first register, and loading the value to return into the second register.

The operations performed by a procedure (or continuation) before it yields the processor to the next procedure are extremely limited. Registers can be loaded with (1) a constant, (2) the original contents of another register, (3) a value retrieved from the procedure's environment, (4) some simple function of values obtained in those ways, or (5) a newly created procedure. Newly created procedures are given environments drawn from the running procedure's environment and the original registers. Each register is only loaded *once* in one of these ways. The resulting set of possible stepping behaviors is not significantly more complex than some machine instruction sets.

To build a tower of processors the user of Stepper needs some way to shift between levels. The two procedures `procedure->implementation` and `implementation->procedure` exist for this purpose. The "implementation" of a procedure is another procedure, which describes the operations to be performed on the registers of the machine when that procedure is invoked. Not surprisingly, this is exactly what the Scheme procedures which represent Stepper procedures do. `Procedure->implementation` can be used within Stepper to single-step a computation, as the following dialog illustrates:

```
-> (set! inc (lambda (n) (+ n 1)))
#<Procedure INC>
-> (inc 3)
4
-> ((procedure->implementation inc) 'k 3)
(#<Eval-Function-Continuation 100553466>
 #<Procedure "+">)
```

The procedure `inc` adds one to its single argument, but the *implementation* of `inc` takes two arguments, a continuation and a number, and returns a tuple representing the next step to perform in evaluating such a call to `inc`. By repeatedly calling implementations, we can step our way through the computation. For example, consider the following useful debugging tool:

```
-> (set! step-until
     (lambda (stop tuple)
       ((lambda (tuple)
          (if (eq? stop (first tuple))
              tuple
              (step-until stop tuple)))
        (apply (procedure->implementation (first tuple))
               (rest tuple)))))
#<Procedure STEP-UNTIL>
```

Step-until takes a tuple representing a computation and a procedure to trap calls to. The computation is stepped until that procedure is about to be invoked, at which point step-until returns the current tuple for examination.

```
-> (set! fact
     (lambda (n)
       (if (< n 2)
           1
           (* (fact (- n 1)) n))))
#<Procedure FACT>
-> (fact 5)
120
-> (step-until * (list fact 'k 5))
(#<Procedure "*"> #<Eval-Arg-Continuation 63365247> 1 2)
```

Here we have advanced the computation of (fact 5) until the very first time the procedure * is invoked. We can continue in this way until the symbol k, which was supplied as the initial continuation, reappears:

```
-> (step-until * %)
(#<Procedure "*"> #<Eval-Arg-Continuation 63365271> 2 3)
-> (step-until * %)
(#<Procedure "*"> #<Eval-Arg-Continuation 63364542> 6 4)
-> (step-until * %)
(#<Procedure "*"> K 24 5)
-> (step-until 'k %)
(K 120)
```

(The variable % is bound to the previous value by the Stepper toplevel loop.)

The procedure implementation->procedure exactly reverses the effect of procedure->implementation. This allows the Stepper programmer to implement procedures by specifying their step-by-step behavior in terms of tuples. For example, here is the definition of call/cc in Stepper:

```
-> (set! call/cc
     (implementation->procedure
       (lambda (k f)
         (list f k (implementation->procedure
                     (lambda (ignored-k v)
                       (list k v)))))))
#<Procedure CALL/CC>
```

The implementation of call/cc specifies precisely how to be step a tuple whose first element is call/cc. The first argument is to be invoked as a procedure with the same continuation as was given to call/cc. It is to be passed a single argument, which is also a procedure whose behavior is specified by giving an implementation. That procedure is to ignore its continuation, and return the value it is given to the original continuation.

An arbitrary procedure can be passed as an implementation to implementation->procedure. The only requirement is that it return a valid tuple (if it returns at all). This implementation procedure will be run on the next level up of the tower of processors we are constructing. By calling implementation->procedure enough times, a procedure can be generated that does its work arbitrarily far up in the tower.

The procedures implementation->procedure and procedure->implementation themselves are not very exciting. They simply tag or untag their argument as appropriate so that other Stepper primitives can recognize them:

```
(define (make-implementation proc)
  (list 'implementation proc))

(define (implementation? imp)
  (and (pair? imp)
       (eq? (car imp) 'implementation)))

(define implementation-procedure cadr)

(define (make-implemented-procedure imp)
  (list 'implemented-procedure imp))

(define (implemented-procedure? proc)
  (and (pair? proc)
       (eq? (car proc) 'implemented-procedure)))

(define procedure-implementation cadr)
```

```
(define (procedure->implementation proc)
  (if (implemented-procedure? proc)
      (procedure-implementation proc)
      (make-implementation proc)))

(define (implementation->procedure imp)
  (if (implementation? imp)
      (implementation-procedure imp)
      (make-implemented-procedure imp)))
```

Toplevel continuations are produced by `make-toplevel-continuation`:

```
(define (make-toplevel-continuation prompt)
  (define (toplevel-continuation val)
    (newline) (write val)
    (newline) (display prompt)
    (evaluate toplevel-continuation
              (read)
              (extend '() '(%) (list val))))
  toplevel-continuation)
```

This is the *only* procedure that calls into the evaluator. All of Stepper could be reconstructed around any other theory of simple transformations on tuples, and only this procedure and the evaluator would need to be changed.

One other continuation is particularly important:

```
(define (omega tuple)
  (list* (procedure->implementation (first tuple))
         omega
         (rest tuple)))
```

When this continuation is given a tuple, it goes into an infinite loop stepping that tuple. All levels of the tower above the first will initially be executing this loop; every level of the tower will be stepping the level below.

Of course there is no need to actually perform an infinite amount of stepping. We really only need to find the highest level that is engaged in some *other* activity, and run it directly. This leads to the following set of tricks:

```
(define (run)
  (find-level-loop
    (list (make-toplevel-continuation "-> ") '*)))
```

This is the procedure used to start Stepper running. It simply calls `find-level-loop` with an initial tuple. Instead of

```
(list (make-toplevel-continuation "-> ") '*)
```

the initial tuple could be

```
(list omega
      (list (make-toplevel-continuation "-> ")
            '*))
```

or even

```
(list omega
      (list omega
            (list (make-toplevel-continuation "-> ")
                  '*)))
```

for in some sense the initial tuple is really the limit of this sequence of tuples. But it doesn't make any practical difference, since `find-level-loop` is only interested in finding the first tuple from the top whose head is something other than `omega`:

```
(define (find-level-loop tuple)
  (let ((head (first tuple)))
    (cond ((eq? head omega)
           (find-level-loop (second tuple)))
          ((implemented-procedure? head)
           (find-level-loop
             (list* (procedure-implementation head)
                    omega
                    (rest tuple))))
          (else
           (find-level-loop
             (step-loop head (rest tuple)))))))
```

Having found a tuple headed by something other than `omega`, `find-level-loop` checks for procedures created by `implementation->procedure`. Such procedures are run one level up in the tower, so `find-level-loop` shifts back up a level and provides a continuation of `omega`. A look at the definition of `omega` will reveal that this is exactly the same behavior as would have resulted if the `omega`-headed tuple at the level above had been run directly.

Finally, having shifted to the level where the action is, `find-level-loop` calls `step-loop`:

```
(define (step-loop head more)
  (if (implementation? head)
      (list (first more)
            (step-loop (implementation-procedure head)
                       (rest more)))
      (apply head more)))
```

`Step-loop` simply produces the next tuple appropriate for the level at which it is invoked. In cases where the head of the current tuple was produced by `procedure->implementation`, `step-loop` recovers the original procedure, steps it once, and then passes the resulting tuple to the continuation. Otherwise `step-loop` simply calls `apply`.

Note that I have not provided each level of the tower with its own toplevel loop. In fact, I have been unable to devise a clean way to do so. I suspect that this is an artifact of the elimination of level-shifting continuations, but I do not understand why this should be the case.

5 Observations

Having now constructed a tower of processors, it is natural to wonder if the result has any intrinsic utility. It wasn't a really a design goal that Stepper be useful for anything; I was only interested in demonstrating that it is possible to construct a tower free from reified expressions and environments, and free of level-shifting continuations. The Stepper tower retains one of the most useful properties of the 3-Lisp tower: the ability to implement transparent debugging utilities. The procedure `step-until`, demonstrated above, is an example of such a utility.

The utility of stepping is enhanced by the ability to define procedures that behave atomically. For example, we can define a version of the factorial function that does its computation one level up in the tower, and works in a single step at the level where it was called:

```
-> (set! atomic-fact
     (implementation->procedure
       (lambda (k n)
         (list k (fact n)))))
#<Procedure ATOMIC-FACT>
-> (atomic-fact 7)
5040
-> ((procedure->implementation fact) 'k 7)
(#<Eval-Function-Continuation 54513442>
 #<Procedure "<">)
-> ((procedure->implementation atomic-fact) 'k 7)
(K 5040)
-> ((procedure->implementation cdr) 'k '(a b c))
(K (B C))
```

`Atomic-fact` is just as good at computing factorials as `fact` was. The difference is that anyone stepping through any code that calls `atomic-fact` will see it do all of its work in a single step, just as if it were a built-in primitive. (Of course one more level up the tower someone stepping through the code that is stepping the call to `atomic-fact` will see `atomic-fact` broken up into more primitive steps.) Thus `implementation->procedure` can be used to provide a kind of limited abstraction; only those steps of a computation that the programmer is interested in need be revealed.

Stepper, as it exists now, lacks effective tools for examining the elements of tuples. `Procedure->implementation` can be used to break a complex procedure down into a series of atomic steps, but no tools are provided for examining those atomic steps in further detail. These atomic steps are simple transformations on tuples, as explained above, so in theory this isn't a hard job. In practice, the design of a description language for atomic steps requires a more detailed construction than is appropriate here.

The techniques employed by `find-level-loop` to implement the Stepper tower are taken more or less directly from the implementation of 3-Lisp given in [1]. In particular, the continuation `omega` is given special-case treatment similar to that given to `normalize` in 3-Lisp. It is interesting that neither version of Brown required such implementation tricks. I suspect that this is because Brown has no effective way of decomposing a procedure, as there are in 3-Lisp and Stepper, and so the only thing that can be done with a procedure is to directly run it. Thus there is no need to ever optimize the indirect execution of a Brown procedure. This makes me further suspect that the denotational semantics developed in [7] might be more difficult if decomposition of Brown procedures was allowed. In fact, I do not see how to construct a denotational description of the Stepper tower.

The most important thing about Stepper is that it demonstrates that the notion of a tower of processors can be separated from the mechanics of an evaluator. Such evaluator artifacts as expressions, environments, and continuations can be confined within the boundaries of single levels, and a much simpler mechanism can be used to manage the interactions between levels. Stepper achieves this separation by choosing a representation for the state of a process (the tuple) in which the entire state is explicitly represented.

In contrast, the state of a 3-Lisp process is partly implicit in the state of the superior 3-Lisp process. The state of a 3-Lisp process can only be understood by knowing exactly *where* in the interpreter the superior process is currently executing. This confusion of levels is what results in level-shifting continuations. By representing all the state of a computation explicitly, Stepper avoids this confusion.

Stepper reveals that once an appropriate representation of processor state has been chosen, the construction of a processor tower is quite straightforward. Processor towers might be easily constructed for a wide variety of systems to enable them to manipulate and reason about themselves. The problem of constructing a well-disciplined tower thus reduces to designing a representation in which the entire state of a process is explicitly represented, which is difficult, and then constructing the tower, which as we have seen is relatively easy.

References

[1] J. des Rivières and B. C. Smith. The implementation of procedurally reflective languages. In *Proc. Symposium on Lisp and Functional Programming*, pages 331–347, ACM, August 1984.

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

[3] Jonathan Rees and William Clinger. *Revised^3 Report on the Algorithmic Language Scheme.* Memo 848a, MIT AI Lab, September 1986.

[4] B. C. Smith. *Reflection and Semantics in a Procedural Language.* TR 272, MIT LCS, January 1982.

[5] B. C. Smith. Reflection and semantics in Lisp. In *Proc. Symposium on Principles of Programming Languages*, pages 23–35, ACM, January 1984.

[6] Guy L. Steele Jr. *LAMBDA: The Ultimate Declarative.* Memo 379, MIT AI Lab, November 1976.

[7] M. Wand and D. P. Friedman. The mystery of the tower revealed: a non-reflective description of the reflective tower. In *Proc. Symposium on Lisp and Functional Programming*, pages 298–307, ACM, August 1986.

Appendix: Stepper

```
; The evaluator

(define (evaluate k e r)
  ((cond ((symbol? e) evaluate-identifier)
         ((not (pair? e)) evaluate-constant)
         (else (case (first e)
                 ((quote) evaluate-quote)
                 ((lambda) evaluate-lambda)
                 ((set!) evaluate-set!)
                 ((if) evaluate-if)
                 (else evaluate-application))))
   k e r))

(define (evaluate-identifier k e r)
  (list k (get-value e r)))

(define (evaluate-constant k e r) (list k e))

(define (evaluate-quote k e r) (list k (second e)))

(define (evaluate-lambda k e r)
  (define (procedure k1 . vals)
    (evaluate k1
              (third e)
              (extend r (second e) vals)))
  (list k procedure))
```

```
(define (evaluate-set! k e r)
  (define (set!-continuation val)
    (set-value (second e) val r)
    (list k val))
  (evaluate set!-continuation (third e) r))

(define (evaluate-if k e r)
  (define (if-continuation val)
    (evaluate k ((if val third fourth) e) r))
  (evaluate if-continuation (second e) r))

(define (evaluate-application k e r)
  (define (eval-function-continuation f)
    (define (loop args vals)
      (define (eval-arg-continuation val)
        (loop (cdr args) (cons val vals)))
      (if (null? args)
          (list* f k (reverse vals))
          (evaluate eval-arg-continuation
                    (first args)
                    r)))
    (loop (rest e) '()))
  (evaluate eval-function-continuation (first e) r))


; Important continuations

(define (make-toplevel-continuation prompt)
  (define (toplevel-continuation val)
    (newline) (write val)
    (newline) (display prompt)
    (evaluate toplevel-continuation
              (read)
              (extend '() '(%) (list val))))
  toplevel-continuation)

(define (omega tuple)
  (list* (procedure->implementation (first tuple))
         omega
         (rest tuple)))
```

```
; Stepper toplevel loop

(define (run)
  (find-level-loop
    (list (make-toplevel-continuation "-> ") '*)))

(define (find-level-loop tuple)
  (let ((head (first tuple)))
    (cond ((eq? head omega)
           (find-level-loop (second tuple)))
          ((implemented-procedure? head)
           (find-level-loop
             (list* (procedure-implementation head)
                    omega
                    (rest tuple))))
          (else
           (find-level-loop
             (step-loop head (rest tuple)))))))

; HEAD is known not to be an implemented procedure
; when STEP-LOOP is called
(define (step-loop head more)
  (if (implementation? head)
      (list (first more)
            (step-loop (implementation-procedure head)
                       (rest more)))
      (apply head more)))
```

```
; Environments

(define global-environment '())

(define (extend r ids vals)
  (cond ((null? ids) r)
        ((not (pair? ids)) (cons (cons ids vals) r))
        (else (extend (cons (cons (car ids)
                                  (car vals))
                            r)
                      (cdr ids)
                      (cdr vals)))))

(define (get-value id r)
  (cdr (get-pair id r)))

(define (set-value id val r)
  (set-cdr! (get-pair id r) val)
  val)

(define (get-pair id r)
  (or (assq id r)
      (assq id global-environment)
      (let ((pair (cons id undefined)))
        (set! global-environment
              (cons pair global-environment))
        pair)))
```

```
; Set up the initial global environment

(define (exportable-apply k cvt-proc . args)
  (list* cvt-proc k (apply list* args)))

(define (convert-procedure f)
  (define (converted-procedure k . args)
    (list k (apply f args)))
  converted-procedure)

(define (initialize-global-environment)
  (set! global-environment
        (append
          (map cons
               '(apply omega)
               (list exportable-apply omega))
          (map (lambda (id proc)
                 (cons id (convert-procedure proc)))
               '(cons list car cdr eq?
                 + - * / = > <
                 first second third fourth rest
                 implementation->procedure
                 procedure->implementation
                 make-toplevel-continuation
                 )
               (list cons list car cdr eq?
                     + - * / = > <
                     first second third fourth rest
                     implementation->procedure
                     procedure->implementation
                     make-toplevel-continuation
                     ))))
 #t)

(initialize-global-environment)
```

```
; Converting between implementations and procedures

(define (make-implementation proc)
  (list 'implementation proc))

(define (implementation? imp)
  (and (pair? imp)
       (eq? (car imp) 'implementation)))

(define implementation-procedure cadr)

(define (make-implemented-procedure imp)
  (list 'implemented-procedure imp))

(define (implemented-procedure? proc)
  (and (pair? proc)
       (eq? (car proc) 'implemented-procedure)))

(define procedure-implementation cadr)

(define (procedure->implementation proc)
  (if (implemented-procedure? proc)
      (procedure-implementation proc)
      (make-implementation proc)))

(define (implementation->procedure imp)
  (if (implementation? imp)
      (implementation-procedure imp)
      (make-implemented-procedure imp)))
```

```
; Various utilities

(define (list* head . tail)
  (if (null? tail)
      head
      (cons head (apply list* tail))))

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

(define undefined (list '* 'undefined '*))
(define (undefined? v) (eq? v undefined))
```