Reification: Reflection without Metaphysics

by

Daniel P. Friedman and Mitchell Wand

Computer Science Department

Indiana University

Bloomington, IN 47405

TECHNICAL REPORT NO. 161
Reification: Reflection without Metaphysics

by

D. P. Friedman and M. Wand

May, 1984

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

Reification: Reflection without Metaphysics

Daniel P. Friedman
Mitchell Wand

Computer Science Department
Indiana University
Lindley Hall 101
Bloomington, IN 47405 USA

Abstract

We consider how the data structures of an interpreter may be made available to the program it is running, and how the program may alter its interpreter's structures. We refer to these processes as *reification* and *reflection*. We show how these processes may be considered as an extension of the fexpr concept in which not only the form and the environment, but also the continuation, are made available to the body of the procedure. We show how such a construct can be used to effectively add an unlimited variety of "special forms" to a very small base language. We consider some trade-offs in how interpreter objects are reified. Our version of this construct is similar to one in 3-Lisp [Smith 82, 84], but is independent of the rest of 3-Lisp. In particular, it does not rely on the notion of a tower of interpreters.

1. Introduction

Fexprs extend the Lisp interpreter by seizing the interpreter data structures and allowing programs to work on them. The basic fexpr mechanism seizes only a portion of the *exp* register (or some similar data structure), but the so-called "2-argument fexpr" gives the program a copy of the environment register as well. `catch` or `call/cc` does the same thing with the continuation register or its equivalent. We refer to constructs of this flavor as *reifiers*.

In this paper, we provide a systematic reification facility in the context of continuation-passing interpreters. In addition, we make these objects first-class citizens of the language. This gives a more orthogonal design. The resulting mini-language is called "Brown."

This material is based on work supported by the National Science Foundation under grant numbers MCS 8303325 and MCS 8304567.

We present a *complete* annotated interpreter for Brown and discuss some of the issues it raises. In Section 2, we present the basic structure of the interpreter. In Section 3, we define reification and reflection and show how we chose to implement those notions. This implementation involved non-trivial design choices. Section 4 concludes the discussion of the interpreter by showing the user interface. Section 5 shows some examples of how this facility can be used to define new special forms. Last, in Section 6, we present some conclusions.

2. The Interpreter

Our interpreter is written in Scheme 84 [Friedman *et al.* 84, Haynes, Friedman, & Wand 84, Section 2] (though T, Common Lisp, etc. would suffice) and is structured much like an ordinary continuation-passing interpreter. The main function, `denotation`, takes an expression and produces a Scheme function which in turn takes an environment and a continuation and produces an answer.

`denotation` dispatches on the syntactic type of the expression and invokes one of a set of "semantic auxiliaries," which take an expression of known syntactic type and produce the appropriate semantic function. We distinguish these auxiliaries by using names which begin with `<` and end with `>`, *e.g.* `<identifier>`. In the code that follows, `e`, `r`, and `k` are expression, environment, and continuation, respectively. A continuation is a function of one argument, as usual, and an environment is a function which takes two arguments, the identifier to be retrieved and the continuation waiting for the L-value associated with that identifier.

```
(define denotation
  (lambda (e)
    (case (syntactic-type e)
      [identifier (<identifier> e)]
      [abstraction
        (let ([b (body e)])
          (<abs>
            (if (reifier? b)
                <reify>
                <simple>)
            (body b)))]
      [application (<app> e)])))
```

The first semantic auxiliary is <identifier>, which consults the environment to find the L-value for the identifier, and passes the associated R-value to its continuation.

```
(define <identifier>
   (lambda (e)
      (lambda (r k)
         (r e
            (lambda (cell)
                 (k (value-of cell)))))))
```

The first difference between our interpreter and a conventional one is that our application line (implemented in the semantic auxiliary <app>) uses call-by-text rather than call-by-value. This choice is forced upon us by the possibility that a function may seize, using a reifier, the text of its call.

```
(define <app>
   (lambda (e)
      (lambda (r k)
         ((denotation (first e))
          r
          (lambda (f)
             (f (rest e) r k))))))
```

This protocol is utilized in the code for abstractions. The interpreter supports two primitive kinds of abstractions: *simple* and *reifying*. Each of these causes its body to be executed in an environment in which the values passed to it are bound to the formal parameters. Thus the underlying function for each abstraction (abs *Type Formals Body*) is of the form

```
(lambda (v* c) ((denotation Body)
                (ext r Formals v*)
                c))
```

where r is the lexical environment. The difference is in what values are passed to this function. In a simple abstraction, the actual parameters are evaluated in the usual way, and the list of values is passed as v*. In a reifying abstraction, what is passed is a list of three elements, consisting of the list of actual parameters, a representation of the current environment, and a representation of the current continuation.

We code this by factoring it into two steps: <abs> produces the underlying function for the abstraction and then passes it to an abstraction-builder (either <simple> or <reify>) to build an appropriate function, which will do the right thing in a call-by-text environment:

```
(define <abs>
   (lambda (abs-builder e)
      (lambda (r k)
         (k (abs-builder
               (lambda (v* c)
                  ((denotation (second e))
                   (ext r (first e) v*)
                   c)))))))

(define <reify>
   (lambda (fun)
      (lambda (e r k)
         (fun
            (list
               e
               (schemeU-to-brown r)
               (schemeK-to-brown k))
            wrong))))

(define <simple>
   (lambda (fun)
      (lambda (e r k)
         ((rec loop
             (lambda (e k)
                (if (null? e) (k nil)
                    ((denotation (first e))
                     r
                     (lambda (v)
                        (loop
                           (rest e)
                           (lambda (w)
                              (k (cons v w)))))))))
          e
          (lambda (v*) (fun v* k))))))
```

An elementary example of a reifying abstraction is quote, which may be defined as:

```
(abs reify (e r k) (k (car e)))
```

This returns the first actual parameter, unevaluated, to the continuation. This differs from the fexpr version (lambda (e r) (car e)) only in that it explicitly returns its value to the continuation k. This definition of quote is basic to the functioning of the rest of the system. For example, we use quoted numbers ('3 instead of 3), to avoid requiring a separate case for numbers in the interpreter. Unlike Smith's system, 'a returns the atom a, not (quote a); we designed the interpreter this way to demonstrate the independence of this mechanism from Smith's other philosophical concerns.

Having reified the interpreter's data structures, what can we do with them? At least one should be able to reinstall them. We refer to this process as *reflection*. We provide two methods of reflection. First, we can re-start the interpreter where it left off by merely invoking a reified continuation, as we did in the code for quote. Alternatively, we may recursively invoke the interpreter through

the function `meaning`, which is bound in the initial environment to

```
(<simple>
   (lambda (v* c)
      (c ((denotation (car v*))
          (brown-to-schemeU (cadr v*))
          (brown-to-schemeK (caddr v*))))))
```

This allows us to control order of evaluation. We can illustrate this idea using the example of `if`, which evaluates two of its three arguments. Assume that `ef` has been defined as a truth-functional version of "if" (that is, it evaluates *all* of its arguments, as in `(define ef (lambda (b x y) (if b x y)))`. Then `if` can be defined as:

```
(abs reify (e r k)
   (meaning (car e) r
      (abs simple (v)
         (meaning
            (ef v
               (car (cdr e))
               (car (cdr (cdr e))))
            r
            k))))
```

In the next section we shall see in more detail how reification and reflection are accomplished.

3. Reification and Reflection

In our discussion of the interpreter, we said that a reifier passes to its body representations of the environment and of the continuation. We did not discuss what a good representation might be, nor did we discuss how the body of a reifier might use those representations to alter the behavior of the interpreter. We must now turn our attention to those issues.

It is convenient to have some terminology. We will use the term *reification* to mean the conversion of an interpreter component into an object which the program can manipulate. One can think of this transformation as converting program (*i.e.* the expression being evaluated) into data. We will use the term *reflection* to mean the operation of taking a program-manipulable value and installing it as a component in the interpreter. This is thus a transformation from data to program.

We need one such transformation for each interpreter component. We use the function `schemeX-to-brown` for reifying from the Scheme representation of component `X` to a Brown value. This corresponds to up-arrow in Smith's work. Conversely, we use the function `brown-to-schemeX` for reflecting from Brown values to Scheme representations of component `X`. This corresponds to Smith's down-arrow. These conversions must satisfy the requirement that the reflection of a reified object must be equal to the original, that is, if x is a schemeX, then `(brown-to-schemeX (schemeX-to-brown x))` = x.

We consider, in turn, this pair of transformations for each needed domain.

The first domain we need to consider is the domain SchemeF of "underlying functions" which are used by both simple and reify abstractions. We represent such functions in Scheme by functions which take two arguments, a list of arguments and a continuation. Since such functions are intended to evaluate their arguments, they convert to simple abstractions in Brown. Thus `schemeF-to-brown` is just the semantic auxiliary `<simple>`. The reflection function is somewhat more involved. It takes a brown function (which uses call-by-text, so it takes a list of unevaluated actuals, an environment, and a continuation), and converts it to a SchemeF, which takes a list of actuals and a continuation. Since Scheme will always evaluate the arguments, we need to turn the values of the arguments back into text for the benefit of the brown function. We do this by wrapping each of them in quotes. Because its arguments are quoted, the brown function will ignore its environment argument.

```
(define schemeF-to-brown <simple>)

(define brown-to-schemeF
   (lambda (bf)
      (lambda (v* c)
         (bf (mapcar wrap/quote v*) initenv c))))
```

We next turn to continuations. The usual representation of a continuation, as supplied by `call/cc` or `catch`, is a function of one argument. Similarly, we choose the brown representation of a continuation to be a simple abstraction of one argument which, when invoked, evaluates its parameter and sends it to the stored continuation. Thus, invocation of such a function acts like a "black hole": it throws away the continuation at the point of call. We will discuss alternative reifications later in the paper.

```
(define schemeK-to-brown
   (lambda (k)
      (lambda (e r k1)
         (if (= (length e) 1)
             ((denotation (first e)) r k)
             (wrong
                (list
                   "schemeK-to-brown: "
                   "wrong number of args "
                   e))))))
```

The corresponding reflection function, `brown-to-schemeK`, is similar to `brown-to-schemeF`, except that we must be more cunning in our choice of continuation argument to `bf`. Since `(schemeK-to-brown k)` ignores its continuation argument, the requirement that `(brown-to-schemeK (schemeK-to-brown k))` = k does not constrain us; we choose it to be `I ( = (lambda (x) x))`, so that reified continuations will return to their caller if they do not invoke a previously reified continuation. (We will show an example of this in Section 5).

```
(define brown-to-schemeK
   (lambda (bf)
      (lambda (v)
         (bf (list (wrap/quote v)) initenv I))))
```

The next domain to be considered is that of environments. Smith reifies environments as data structures. We reify them as simple abstractions of one argument, in keeping with the usual practice in semantics. The reified environment takes one argument, an identifier, and returns the associated L-value.

Since any simple abstraction of one argument might be reflected to become an environment, an environment might do an arbitrary amount of computation while looking up an identifier. Hence we cannot represent environments as data structures in the usual way. We must instead represent them as functions.

We represent an environment as a Scheme function which takes an identifier and a continuation, and which passes the associated L-value to that continuation. The function `ext` builds such functions in place of the usual rib-cage or a-list extension; it also checks to see that argument and variable lists are the same length and balks if they are not. The reflection function converts a brown function to a corresponding Scheme function in the usual way.

```
(define schemeU-to-brown
   (lambda (r)
      (lambda (e r1 k1)
         (if (= (length e) 1)
             ((denotation (first e))
              r1
              (lambda (v) (r v k1)))
             (wrong
                (list
                   "schemeU-to-brown: "
                   "wrong no of args "
                   e))))))
```

```
(define brown-to-schemeU
   (lambda (bf)
      (lambda (v c)
         (bf (wrap/quote v) initenv c))))
```

```
(define nullenv
   (lambda (v c)
      (wrong (list "brown: unbound id " v))))
```

```
(define ext
   (lambda (r vars vals)
      (if (= (length vars) (length vals))
          (lambda (v c)
             ((rec lookup
                (lambda (vars vals)
                (cond [(null? vars) (r v c)]
                      [(eq? (first vars) v)
                       (c vals)]
                      [t (lookup
                            (rest vars)
                            (rest vals))])))
              vars vals))
          (begin
             (writeln
                "Brown: wrong number of actuals")
             (writeln "Formals: " vars)
             (writeln "Actuals: " vals)
             (wrong "ext failed")))))
```

4. The Initial Environment

To make this interpreter usable, we need to set up an initial environment and an initial continuation. In the initial environment, we provide a small set of initial functions from Scheme, converted to SchemeF's. We package the data manipulation functions so that the programmer can operate entirely in Brown. Thus in the initial environment, `ext` is bound to

```
(lambda (br p* v*)
   (schemeU-to-brown
      (ext (brown-to-schemeU br) p* v*)))
```

The complete code for this is shown in the appendix. Last, we need to set up an initial continuation. This is done by seizing a Scheme continuation and using it as a SchemeK. We also define `wrong` as a SchemeK:

```
(define run
   (lambda (e)
      (call/cc
         (lambda (caller)
            ((denotation e) initenv caller)))))
```

```
(define wrong
   (lambda (v)
      (writeln "wrong: " v) (reset)))
```

5. Defining Special Forms

Just as the Fexpr facility may be thought of as defining new special forms in the Lisp interpreter, a reifier may be thought of as defining a new special form in Brown. We have already shown the examples of `quote` and `if` above. Another standard example is `call/cc`, which invokes its argument on the current continuation:

```
(define-brown call/cc
    (abs simple (f)
       ((abs reify (e r k) (k (f k))))))
```

A macro facility can be defined as follows:

```
(define-brown macro
   (abs simple (bf)
      (abs reify (e r k)
         (meaning (bf e) r k))))
```

`(macro` *bf*`)` creates a reifying abstraction which, when invoked, applies *bf* to its reified application; the resulting expression is then evaluated. As an example, we write `lambda` as a macro. Here `(lambda` *vars body*`)` expands to `(abs simple` *vars body*`)`; in the code for the macro, `e` becomes bound to `(`*vars body*`)`:

```
(define-brown lambda
   (macro
      (abs simple (e)
         (cons 'abs (cons 'simple e)))))
```

Another illustration of the use of reifiers to control the order and extent of evaluation is `set!`. `set!` evaluates its second argument and then updates the store in the L-value associated with its unevaluated first argument:

```
(define-brown set!
   (abs reify (e r k)
      (meaning (car (cdr e)) r
         (abs simple (v)
            (k (update-store
                  (r (car e))
                  v))))))
```

Another useful special form is `begin` (*i.e.* `progn`), which takes a series of expressions and evaluates them from left to right, returning the value of the last one. We will include several versions. The first is the simplest:

```
(define-brown begin
   (abs reify (e r k)
      ((abs simple (dummy)
          (meaning (car (cdr e)) r k))
       (meaning (car e) r (abs simple (x) x)))))
```

This version takes two expressions, evaluates the first with the identity continuation, and evaluates the second, throwing away the value of the first. This illustrates our choice of the continuation `(lambda (x) x)` in `brown-to-schemeK` (other feasible choices, such as `wrong`, would cause this program to fail). It is, however, rather contrary to the spirit of denotational semantics, which would prefer a version in which all continuations were tail recursive. That more standard version is:

```
(define-brown begin
   (abs reify (e r k)
      (meaning (car e) r
         (abs simple (v)
            (meaning (car (cdr e)) r k)))))
```

Our third and last version of `begin` takes an arbitrary number of expressions. It uses an applicative-order fixpoint operator `fix1` [Steele & Sussman 78, p. 70] to construct a local loop for evaluating the expressions.

```
(define-brown fix1
   (abs simple (f)
      ((abs simple (d) (d d))
       (abs simple (g)
          (abs simple (x) ((f (g g)) x))))))

(define-brown begin
   (abs reify (e r k)
      ((fix1
          (abs simple (loop)
             (abs simple (e)
                (if (null? (cdr e))
                    (meaning (car e) r k)
                    (meaning (car e) r
                       (abs simple (v)
                          (loop (cdr e))))))))
       e)))
```

With some additional programming, one can similarly define a lexpr facility:

```
(define-brown lexpr
   (abs reify (e r k)
      (k (abs reify (e1 r1 k1)
            (meaning* e1 r1
               (abs simple (v*)
                  (meaning
                     (car (cdr e))
                     (ext r (car e) (cons v* nil))
                     k1)))))))
```

where meaning* takes a list of expressions and computes the list of their values. Thus one could say:

```
(define-brown list (lexpr (v) v))
```

An example which illustrates our choice of reifications is attach. (attach id body) evaluates body in an environment with the property that whenever identifier id is probed, an informative message is produced. This effect is more difficult to achieve using Smith's reification of environments as list structures.

```
(define-brown attach
   (abs reify (e r k)
      (meaning
         (car (cdr e))
         (abs simple (v)
            (begin
               (if (eq? v (car e))
                   (print "Probed!")
                   nil)
               (r v)))
         k)))
```

6. Conclusions

Our goal in this work was to attempt to understand the idea of reflection, as advocated by Smith, in a way which was motivated by our understanding of semantics, and without being overly concerned with overriding philosophical issues. For us, reification is just a way of giving the interpreter's data to the program, and reflection is a way of loading the interpreter's registers from the program. From this point of view, we can begin to explore alternative reification/reflection pairs:

1. We chose to reify environments as functions, rather than the data structures Smith used. We chose functions because they are closer to the traditional semantic treatment of environments. Data structures permit the program to explore and modify the rib or a-list structure (for better or worse). Functions encapsulate the rib structure, but make it easier to make procedural attachments to variables (*e.g.* attach).

2. We chose to invoke the body of a reifier with the continuation wrong, so that it was an error to fall off the end. We could have gotten behavior more like that of 3-Lisp if we had chosen to use a read/eval/print loop instead.

3. We chose to make (lambda (x) x) the continuation parameter in brown-to-schemeK. One gets different behavior by using, say, wrong instead.

4. We chose a black-hole representation for schemeK-to-brown. If one invokes a reified continuation one loses the current context, just as invoking a continuation is an unconditional goto in Scheme. 3-Lisp uses a different reification, in which a continuation returns to the place it is invoked. This can be done, at the expense of making the invocation non tail-recursive.

5. We chose to reify the denotation function as a simple function which returns to its caller when its continuation returns. (See the first version of begin). We could then trick this mechanism into failing to return by passing it a continuation which did not return, either by concluding with a (reset), a Scheme continuation, or a read-loop. Another approach would be to reify the meaning function as

   ```
   (<simple>
      (lambda (v* c)
         ((denotation (car v*))
          (brown-to-schemeU (cadr v*))
          (brown-to-schemeK (caddr v*)))))
   ```

   This differs from the present one in that the current continuation c is ignored; the three arguments are effectively reinstalled in the registers of the interpreter, replacing the ones that were there. This function might be called reflect; it exposes the fact that denotation is a reflection operation like brown-to-schemeU and brown-to-schemeK. It is somewhat more palatable semantically but is somewhat less tower-like, illustrating that the idea of reflection is independent of Smith's notion of towers of interpreters.

One can ask the same question of a reification facility that one often asks of the fexpr facility: Why is this better than macros? The answer is that one can do some things that one can't do with macros, such as attaching procedures to identifier accesses in environments. One pays the price, however, of essentially keeping the compiler (that is, the function denotation) resident at all times, whereas macros are expanded once, at compile time, and need not be

resident at run time. A possible compromise lies in recognising that macros are essentially a compile-time reification facility. We are currently exploring the implications of that idea.

One cannot forever ignore the philosophical problems (nominalism *vs.* realism, reduction *vs.* evaluation) raised by Smith's treatment of reflection. From the Brown experience, we conclude that reflection *per se* is independent of these issues, as it is independent of the notion of towers of interpreters. We hope to report on these other aspects elsewhere.

References

[Friedman *et al.* 84]
Friedman, D.P., Haynes, C.T., Kohlbecker, E., and Wand, M. "The Scheme 84 Reference Manual" Indiana University Computer Science Department Technical Report No. 153 (March, 1984).

[Haynes, Friedman, & Wand 84]
Haynes, C.T., Friedman, D.P., and Wand, M., "Continuations and Coroutines," *Proc. 1984 ACM Symp. on LISP and Functional Programming.*

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

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

[Steele & Sussman 78]
Steele, G.L. Jr. and Sussman, G.J. "The Art of the Interpreter or, the Modularity Complex (Parts Zero, One and Two)," Mass. Inst. of Tech. Artif. Intell. Memo No. 453, Cambridge, MA (May, 1978).

Appendix – Help Functions

```
; tagging functions and aliases

(define tagit cons)
(define tag car)
(define body cdr)
(define value-of car)
(define first car)
(define second cadr)
(define rest cdr)

(define wrap/quote (lambda (v) (list 'quote v)))

(define I (lambda (x) x))

; syntactic auxiliaries

(define abs?
   (lambda (f) (eq? (tag f) 'abs)))
(define reifier?
   (lambda (f) (eq? (tag f) 'reify)))
(define simple?
   (lambda (f) (eq? (tag f) 'simple)))

(define syntactic-type
   (lambda (e)
      (cond
         ((atom? e) 'identifier)
         ((abs? e) 'abstraction)
         (t 'application))))

; ** the interpreter (Sections 2-3) goes here **

; auxiliaries for setting up initenv

; convert direct scheme fcns to schemeF

(define scheme-to-schemeF

   (lambda (f)
      (lambda (v* c)(c (apply f v*)))))

(define define-brown1
   (lambda (name exp)
      (call/cc
         (lambda (caller)
            ((denotation exp)
             initenv
             (lambda (v)
                (set! initenv
                   (ext initenv
                      (list name)
                      (list v)))
                (caller name)))))))

; define-brown is a useful macro

(mkmac (define-brown id val)
       (define-brown1 'id 'val))

; ** the top level (Section 4) goes here **
```

```
(define boot-initenv
   (lambda ()
      (let
         ((scheme-fn-table
             (list
                (cons 'car car)
                (cons 'cdr cdr)
                (cons 'cons cons)
                (cons 'eq? eq?)
                (cons 'atom? atom?)
                (cons 'null? null?)
                (cons 'add1 add1)
                (cons 'sub1 sub1)
                (cons '=0 =0)
                (cons '+ +)
                (cons '- -)
                (cons '* *)
                (cons 'print print)
                (cons 'length length)
                (cons 'read read)
                (cons 'ext
                   (lambda (br p* v*)
                      (schemeU-to-brown
                         (ext
                            (brown-to-schemeU br)
                            p*
                            v*))))
                (cons 'nullenv nullenv)
                (cons 'update-store
                   (lambda (x y)
                      (value-of (set-car! x y))))
                (cons 'reifier? reifier?)
                (cons 'simple? simple?)
                (cons 'abs? abs?)
                (cons 'wrong wrong)
                (cons 'ef
                   (lambda (bool x y) (if bool x y)))
                (cons 'newline newline)
                (cons 'meaning
                   (lambda (e r k)
                      ((denotation e)
                       (brown-to-schemeU r)
                       (brown-to-schemeK k)))))))
          (let ((initvars (mapcar car scheme-fn-table))
                (initvals
                   (mapcar
                      (lambda (x)
                         (schemeF-to-brown
                            (scheme-to-schemeF (cdr x))))
                      scheme-fn-table)))
             (define initenv
                (ext (ext nullenv '(nil t) '(nil t))
                     initvars
                     initvals))
             (define-brown quote
                (abs reify (e r k) (k (car e))))))))

(boot-initenv)
```