Reflecting on the Metalevel Interpreter Written in Direct Style

Kenichi Asai
Department of Information Science, Faculty of Science, Ochanomizu University
2-1-1 Otsuka Bunkyo-ku Tokyo 112-8610 Japan
asai@is.ocha.ac.jp

ABSTRACT

This paper presents a reflective Scheme system in which user programs are allowed to go up to the metalevel interpreter, access to it, and even modify (parts of) it. Since the metalevel interpreter defines the operational semantics of the baselevel program, the system effectively allows users to modify the semantics of the language. The novel characteristic of the system is that the metalevel interpreter is written in direct style (DS). Conventionally, the metalevel interpreter of a system with a (behavioral) reflective facility has typically been written in continuation-passing style (CPS) to make the control structure explicit. In our system, the same information is obtained via delimited continuation constructs: shift and reset, a variant of call/cc. With the help of shift and reset, the metalevel interpreter becomes simpler and easy to understand without sacrificing the reflective abilities.

1. INTRODUCTION

A reflective language enables us to access, modify, and extend the language semantics from within the same language framework. When one wants to modify parts of the language semantics in a conventional non-reflective language, he typically has to go into the implementation of that language written in a low-level language and change it carefully. By using a reflective language, it becomes possible to perform such modification without going into the low-level implementation but in the same high-level language framework that he usually programs in.

A reflective language provides the language semantics in the form of a meta-circular interpreter. User programs (running at the *baselevel* or level 0) are supposed to be executed by the meta-circular interpreter running at the level above (*metalevel* or level 1). User programs are then allowed to go up to the metalevel, access the meta-circular interpreter, or even modify it. The fact that the meta-circular interpreter is written in the same language as it defines raises a question how it is executed. In a typical reflective system, it is executed by another meta-circular interpreter running at level 2, and so on. This leads to an infinite sequence of interpreters, called a *reflective tower*.

The conventional reflective systems, such as 3-LISP [11], Brown [6, 13], and Blond [4], have been typically written in a so-called continuation-passing style (CPS). One of the reasons for this choice is that CPS makes the control structure explicit. Suspending and resuming computation at an arbitrary point is essential in designing a reflective system. It also fits well to the philosophical background that reflective languages allow modification of the language semantics which is typically written in CPS.

However, once a reflective system is constructed and user programs are allowed to access and modify the metalevel interpreter, it is not always desirable that the metalevel interpreter is written in CPS. Rather, it becomes easier to access, modify, and extend the interpreter if it is written in a simpler direct style (DS).

In this paper, we investigate how to write a reflective system in which the user-observable metalevel interpreter is written in DS. We first transform the conventional reflective system written in CPS into DS by *currying* the continuation argument of a CPS interpreter. The resulting system is still written in CPS, but the user-observable metalevel interpreter becomes DS. We further transform it so that the system itself is written in DS. The main tool we use is the control operators `shift` and `reset` proposed by Danvy and Filinski [3]. We show how to encode shifting up and down the level in terms of these operations.

This paper is organized as follows. After presenting prerequisites and notation used in the paper in Section 2, we review our reflective language Black [2] in Section 3, by showing some example execution, the user-observable metalevel interpreter, and the actual implementation. In Section 4, we describe a variant of Black in which the metalevel interpreter is written in DS, but the system itself is written in CPS. In Sections 5 and 6, we further modify the system so that both the metalevel interpreter and the actual implementation are in DS. Related work is in Section 7 and the paper concludes in Section 8. The Appendix A shows the definition of environment manipulating functions used in the paper.

2. PREREQUISITES AND NOTATION

The base language we use is Scheme [9], extended with lazy lists and the control operators `shift` and `reset`. We use `cons-stream`, `head`, and `tail` to construct a lazy list, to take the first element, and to take the rest of the list,

Presented at the International Lisp Conference 2003 (ILC 2003), New York City, October 12-15, 2003.

respectively.

The control operators `shift` and `reset` are introduced by Danvy and Filinski [3] to capture a *part* of a continuation, or a *delimited* continuation. Like `call/cc` in Scheme, `shift` grabs the current continuation. Unlike `call/cc`, however, it grabs the continuation only up to the enclosing reset. For example, in the following expression:

```
(+ 1 (reset (+ 10 (shift k (k (k 100))))))
```

the continuation grabbed by `shift` is `(+ 10 [])`, which is then bound to `k`. The grabbed continuation can be composed. In the above example, it is applied twice yielding 120. This becomes the value of `(reset ...)` and the final answer becomes 121. Unlike `call/cc`, the grabbed continuation is discarded if it is not explicitly used. We assume that `shift` and `reset` are available in the underlying Scheme system. For their implementation using `call/cc` and `set!`, see [10].

In the paper, levelshifting is abstracted by two operators, `shift-up` and `shift-down`. Readers should not confuse them with the control operator `shift`. How to implement the former two operators in terms of the latter is one of the main topics of the paper.

In the Scheme code, we use the variables `e`, `r`, and `k` for an expression, an environment, and a continuation, respectively. For the environment and continuation of the metalevel, we use with a prefix `m`, like `mr` and `mk`. We also refer to a *metacontinuation*. We represent it using a variable `Mcont`.

There appear several similarly-named functions: `scheme-apply`, `base-apply`, and `meta-apply`. The first one is the standard `apply` function available in Scheme and defined as follows:

```
(define scheme-apply apply)
```

We use `scheme-apply` rather than the original `apply` to clearly distinguish it form `base-apply` and `meta-apply`. The latter two are described in the paper in detail.

For the similar reason, the main function of the metalevel interpreter is named `base-eval`, rather than `eval` which is already defined in the most Scheme systems.

3. CPS BLACK IN CPS

The reflective language we consider in this paper is called Black [2]. It is a Scheme-based language with one additional construct `exit`. Unlike the standard Scheme system where `exit` terminates the Scheme interpreter, `(exit` *e*`)` in Black terminates only the current level of the interpreter and returns *e* to the metalevel interpreter.

3.1 Example Execution

To see how `exit` is used, we show some transcripts of the example interaction in the Black system. Black is launched by typing `(black)` to the Scheme prompt.

```
> (black)
0-0: start
0-1> (car (cons 1 2))
0-1: 1
0-2>
```

The Black prompt consists of two figures. The first figure indicates the level in which input expressions are evaluated. In the above transcript, the current level is 0. The second figure shows the number of iterations in the level. In the above transcript, the user inputs `(car (cons 1 2))` at the first iteration, and the Black system is waiting for the second input. When `exit` is not used, Black behaves the same as Scheme. We now use `exit` to exit the current level.

```
0-2> (exit 0)
New level loaded.
1-0: 0
1-1>
```

Whenever a new level is visited for the first time, the Black system prints the message "`New level loaded.`" to emphasize that a new level is (lazily) created. After exiting the original level, the prompt says that we are now at level 1. This is where the metalevel interpreter resides which has been evaluating the level 0 expressions until `exit` is executed. We can now see the metalevel interpreter. The main function of the interpreter is called `base-eval`.

```
1-1> base-eval
1-1: #[procedure base-eval]
1-2>
```

Since the metalevel interpreter is provided as an ordinary function, users are allowed to use, or even modify it. To do it, however, we need to know how the default metalevel interpreter is constructed. In the original Black, it is a standard CPS interpreter written in Scheme. Figure 1 shows the main parts of it.

The main function `base-eval` accepts an expression `e`, an environment `r`, and a continuation `k`, and evaluates `e` under `r` and passes the result to `k`. According to the syntactic class of `e`, `base-eval` dispatches to other `eval-*` functions. In the following, we refer to these functions *evaluator functions*, since they comprise a metalevel evaluator (or an interpreter). To manipulate an environment, helper functions `get`, `extend`, and `set-value!` are used, which takes a binding, extends an environment with a new binding, and destructively updates a binding, respectively. They are all defined in Appendix A.

It is interesting to see `eval-exit` closely. It stores the current environment `r` and continuation `k` into global variables `old-env` and `old-cont`, respectively, and simply returns `(car (cdr e))` discarding `k`. Namely, it quits the current level and returns `(car (cdr e))` to the metalevel. (We could write the interpreter so that `(car (cdr e))` is evaluated before passed to the metalevel.) The other parts of the interpreter are standard. The use of `shift` and `reset` in `eval-lambda` and `base-apply` will be explained later.

Let us go back to the example session. The evaluator functions are available to users as ordinary functions.

```
1-2> (base-eval '(+ 1 2) init-env (lambda (x) x))
1-2: 3
1-3>
```

`init-env` is the initial environment which holds the bindings for predefined symbols, such as `+`. We can even modify the interpreter.

```
1-3> (define old-eval base-eval)
1-3: old-eval
1-4> (set! base-eval
         (lambda (e r k)
```

```
(define (base-eval e r k)
  (cond ((number? e)          (k e))
        ((boolean? e)         (k e))
        ((symbol? e)          (eval-var e r k))
        ((eq? (car e) ’if)    (eval-if e r k))
        ((eq? (car e) ’set!)  (eval-set! e r k))
        ((eq? (car e) ’lambda)(eval-lambda e r k))
        ((eq? (car e) ’exit)  (eval-exit e r k))
        ...
        (else (eval-application e r k))))
(define (eval-var e r k) (k (cdr (get e r))))
(define (eval-if e r k) ; (if pred then else)
  (base-eval (car (cdr e)) r (lambda (p)
    (if p (base-eval (car (cdr (cdr e))) r k)
          (base-eval (car (cdr (cdr (cdr e))))
                     r k)))))
(define (eval-set! e r k) ; (set! var exp)
  (base-eval (car (cdr (cdr e))) r (lambda (data)
    (set-value! (car (cdr e)) data r)
    (k (car (cdr e)))))))
(define (eval-lambda e r k) ; (lambda params body)
  (let ((body (car (cdr (cdr e))))
        (params (car (cdr e))))
    (k (lambda operand (shift k2
         (base-eval body (extend r params operand)
                    k2))))))
(define old-env 0)
(define old-cont 0)
(define (eval-exit e r k) ; (exit exp)
  (set! old-env r)
  (set! old-cont k)
  (car (cdr e)))
(define (eval-application e r k) ; (f a b c ...)
  (eval-list e r (lambda (l)
    (base-apply (car l) (cdr l) r k))))
(define (eval-list e r k)
  (if (null? e) (k ’())
    (base-eval (car e) r (lambda (u)
      (eval-list (cdr e) r (lambda (v)
        (k (cons u v))))))))
(define (base-apply operator operand r k)
  (reset (k (scheme-apply operator operand))))
```

Figure 1: Metalevel CPS interpreter observed by user programs

```
             (write e) (newline) (old-eval e r k)))
1-4: base-eval
1-5>
```

We store the original definition of `base-eval` in `old-eval`[1] and redefine `base-eval` so that it prints `e` to be evaluated before actually evaluating it. To see its effect, we go back to the baselevel using `old-cont` in which the baselevel continuation is stored.

```
1-5> (old-cont 1)
0-2: 1
0-3> (car (cons 1 2))
(car (cons 1 2))
car
(cons 1 2)
cons
1
2
0-3: 1
0-4>
```

After going back to level 0, we can see that the expression `(car (cons 1 2))` is actually evaluated under the redefined semantics. Let us exit level 0 again.

```
0-4> (exit 0)
(exit 0)
1-5: 0
1-6> old-eval
1-6: #[procedure base-eval]
1-7> base-eval
1-7: #[procedure 0x13B7334]
1-8>
```

Since level 0 has been already created before, the message "`New level loaded.`" is not shown here. Because expressions at level 1 are evaluated using the metametalevel (default) interpreter, traces are no longer displayed. It can be observed by exiting once more.

```
1-8> (exit 0)
New level loaded.
2-0: 0
2-1> base-eval
2-1: #[procedure base-eval]
2-2>
```

3.2 Passing Continuations Using Shift/Reset

In `eval-lambda` and `base-apply` in Figure 1, `shift` and `reset` are used to pass the continuation of a function call. In `base-apply`, the result of applying `operator` to `operand` is passed to the continuation `k` in DS, so that the result of addition, for example, is passed to the continuation. If `operator` is a lambda abstraction, however, we need this current continuation `k` to evaluate the body of the abstraction. To obtain it, we enclose the application of `k` with `reset` and use `shift` to grab it in `eval-lambda`.

It is tempting to pass the continuation as an extra parameter as in:

```
((scheme-apply operator operand) k)
```

in `base-apply` and to take it in `eval-lambda` as follows:

---

[1] We use here the special form `define` whose definition is omitted in Figure 1. It can be added easily.

```
(lambda operand (lambda (k)
  (base-eval body (extend ...) k)))
```

This is not acceptable, however. Remember that evaluator functions are available to user programs as ordinary functions. Consider the evaluation of the following expression:

```
(base-eval '(+ 1 2) init-env (lambda (x) x))
```

The result of applying `base-eval` will be 3, but it would then be applied to the current continuation `k`, if `base-apply` were defined as

```
((scheme-apply operator operand) k)
```

Since evaluator functions are observable from user programs, the functionality (type) of evaluator functions and user-defined functions must be identical. To achieve it, we used `shift` and `reset`. This complication does not arise, if we represent a function as a closure.

3.3 Shifting Levels

There are several approaches to implement a reflective system. The simplest approach is to make the interpreter meta-circular (by supporting all the constructs used in the interpreter, such as `cond`, `let`, etc.) and runs it on top of itself [8]. From construction, it allows user programs arbitrary operations on the metalevel interpreter, including modification. This approach, however, severely suffers from the efficiency problem. Introduction of one more level results in an order of magnitude slowdown. Furthermore, the number of levels has to be fixed before user programs are executed.

Another approach is to give up changeability of interpreters in favor of efficiency [4, 6, 13]. In this approach, the metalevel interpreter is hardwired and not allowed to modify. Since the interpreter is immutable, it can be compiled and efficiently executed (rather than interpreted). The reflective power comes from the access to the metalevel information, such as environments and continuations, and their use with a user-defined interpreter.

The approach we took for Black is a hybrid of both the approaches [2]. We execute the metalevel interpreter efficiently without interpretation, while carefully keeping redefinability of metalevel interpreters.

To keep track of the state of the directly executed metalevel interpreters, we use a *metacontinuation* [4, 6, 13]. It is characterized by the following equation:

$$Mcont = (Env \times Cont) \times Mcont \ .$$

A metacontinuation holds the environment and the continuation of the current level as well as of all the upper levels. In the implementation of Black, a metacontinuation is single-threaded and passed around to all the evaluator functions in a way similarly to the store-passing style. Thus, the type of evaluator functions becomes

$$(Exp \times Env \times Cont) \to Mcont \to Ans \ .$$

Although a metacontinuation is passed around, it will not appear usually, since it is η-reduced. Only when levelshifting occurs does it appear explicitly in the implementation. (Furthermore, since we will encapsulate levelshifting in two operations `shift-up` and `shift-down` defined below, a metacontinuation appears only in these functions.)

Since the number of interpreters in the reflective tower is unknown, we implement a metacontinuation using a lazy list. The following two functions are used to shift levels.

```
(define (shift-up code)
  (lambda (Mcont)
    (let ((mr (car (head Mcont)))
          (mk (car (cdr (head Mcont))))
          (meta-Mcont (tail Mcont)))
      ((code mr mk) meta-Mcont))))
(define (shift-down code r k)
  (lambda (Mcont)
    (code (cons-stream (list r k) Mcont))))
```

`(shift-up code)` is used to execute `code` at the metalevel. It extracts a metalevel environment and continuation from the metacontinuation, and evaluates `code` with the tail of `Mcont`, *i.e.*, at the metalevel. Since the metalevel environment `mr` and continuation `mk` is passed to `code`, they are available in `code`. To go down a level, `shift-down` receives code to be executed as well as the current environment `r` and continuation `k`. It then stores `r` and `k` into the metacontinuation and execute `code` at the level below.

`Shift-up` and `shift-down` are inverse of each other. If `mr` and `mk` do not occur free in `code`, then both the following expressions behave the same as `code` if a metacontinuation is passed:

```
(shift-up (lambda (mr mk) (shift-down code mr mk)))
(shift-down (shift-up (lambda (mr mk) code)) mr mk)
```

3.4 The Actual Black Interpreter

With `shift-up` and `shift-down`, we can now implement Black. The key technique to keep redefinability of metalevel interpreters is to *hook* a function call using `meta-apply`. Figure 2 shows the core parts of the implementation of Black written in CPS.

The implementation is almost the same as the metalevel interpreter observed by user programs (Figure 1) except for a few points. First, all the calls to evaluator functions are done through `meta-apply`, whose definition is found at the bottom of Figure 2. It goes up a level, grabs the metalevel environment, takes the bindings for the called function name, and executes it at the original level. In other words, it hooks calls to evaluator functions and checks bindings for the called functions before actually calling them.

The metalevel environment initially holds the default evaluator functions. So, if user programs do not redefine the metalevel interpreter, then the behavior of the Black system is the same as the one without `meta-apply`. If user programs redefine the metalevel interpreter, the new redefined function is stored in the metalevel environment `mr`. When it is called, it is extracted from `mr` in `meta-apply` and executed, achieving the replacement of evaluator functions.

The second point to notice about the implementation of Black is how `eval-exit` is written. In the previous interpreter (Figure 1), `eval-exit` performs two assignments and returns the argument unevaluated. In the reflective tower, these assignments become updates of the metalevel environment `mr`. Thus, in Figure 2, `eval-exit` shifts up one level and updates `mr`. Returning a value to the metalevel is implemented as passing the value to the metalevel continuation `mk`.

Finally, `eval-lambda` and `base-apply`[2] require explana-

---

[2] `(get-global-env r)` appearing in `base-apply` extracts the

```
(define (base-eval e r k)
  (cond ((number? e)  (k e))
        ((boolean? e) (k e))
        ((symbol? e)
         (meta-apply 'eval-var e r k))
        ((eq? (car e) 'if)
         (meta-apply 'eval-if e r k))
        ((eq? (car e) 'set!)
         (meta-apply 'eval-set! e r k))
        ((eq? (car e) 'lambda)
         (meta-apply 'eval-lambda e r k))
        ((eq? (car e) 'exit)
         (meta-apply 'eval-exit e r k))
        ...
        (else
         (meta-apply 'eval-application e r k))))
(define (eval-var e r k) (k (cdr (get e r))))
(define (eval-if e r k) ; (if pred then else)
  (meta-apply 'base-eval (car (cdr e)) r
    (lambda (p)
      (if p (meta-apply
              'base-eval (car (cdr (cdr e))) r k)
            (meta-apply
              'base-eval (car (cdr (cdr (cdr e))))
                   r k)))))
(define (eval-set! e r k) ; (set! var exp)
  (meta-apply 'base-eval (car (cdr (cdr e))) r
    (lambda (data)
      (set-value! (car (cdr e)) data r)
      (k (car (cdr e))))))
(define (eval-lambda e r k) ; (lambda params body)
  (let ((body (car (cdr (cdr e))))
        (params (car (cdr e))))
    (k (lambda operand
         (shift-up (lambda (mr mk)
           (meta-apply 'base-eval body
                       (extend r params operand)
                       mk)))))))
(define (eval-exit e r k) ; (exit exp)
  (shift-up (lambda (mr mk)
    (set-value! 'old-env r mr)
    (set-value! 'old-cont k mr)
    (mk (car (cdr e))))))
(define (eval-application e r k) ; (f a b c ...)
  (meta-apply 'eval-list e r (lambda (l)
    (meta-apply 'base-apply (car l) (cdr l) r k))))
(define (eval-list e r k)
  (if (null? e) (k '())
    (meta-apply 'base-eval (car e) r (lambda (u)
      (meta-apply 'eval-list (cdr e) r (lambda (v)
        (k (cons u v))))))))
(define (base-apply operator operand r k)
  (shift-down (scheme-apply operator operand)
              (get-global-env r) k))
(define (meta-apply proc-name . operand)
  (shift-up (lambda (mr mk)
    (shift-down
      (scheme-apply (cdr (get proc-name mr))
                    operand)
     mr mk))))
```

Figure 2: Implementation of CPS Black in CPS

tion, because they are written using `shift-up` and `-down`. At first sight, they do not seem to require levelshifting, because they are not particular to reflective interpreters but appear in the standard non-reflective interpreters. However, they do require levelshifting. To understand why, we need to reconsider what the levels are.

User programs observe levels via the number of interpreters executed in the reflective tower. For example, if

```
(base-eval '(+ 1 2) init-env (lambda (x) x))
```

is executed, then a new level is launched and `'(+ 1 2)` is executed at the level below. In the case of `'(+ 1 2)`, it dose not make any difference whichever level it is executed, but if we replace the expression with a read-eval-print loop, then we want to execute it at the level below. This is why `base-apply` requires shifting down. A new level is launched when evaluator functions are executed.

The shifting down in `base-apply` has several consequences to the design of other parts of the interpreter. First, `eval-lambda` has to return a closure which, when applied, evaluates its body at the level above, canceling the shifting down in `base-apply`. Secondly, all the primitive functions have to be rewritten so that they return their results to the metalevel continuation. For example, `+` is bound to

```
(lambda operand
  (shift-up (lambda (mr mk)
    (mk (scheme-apply + operand)))))
```

Readers might think that this way of shifting down in `base-apply` makes it impossible to come back to the current level after the execution of evaluator functions. This is not the case because eventually the continuation argument is applied. In the above example, the continuation `(lambda (x) x)` will be applied after the execution of `'(+ 1 2)`, where the original level is restored through shifting up in the closure created for `(lambda (x) x)`.

The shifting down and up in `base-apply` and `eval-lambda` fit nicely to the redefinition of a part of the metalevel interpreter. Suppose, for example, that `eval-if` in the metalevel interpreter is redefined to a user-define function. When the default `base-eval` calls the redefined `eval-if`, the level shifts up to the metametalevel (because of the shifting up in `eval-lambda` in the metametalevel) to evaluate the body of new `eval-if`. In its body, `eval-if` typically calls `base-eval`. At this time, the level shifts down to the metalevel because of the shifting down in `base-apply` in the metametalevel, and the evaluation continues at the metalevel. With the shifting down and up in `base-apply` and `eval-lambda`, default evaluator functions and user-defined functions co-exist at the same level. This is called *linguistic symbiosis* in the literature [7].

Notice that even though the Black system is implemented like this, the metalevel interpreter observed by user programs remains the same as before (Figure 1). User programs do not see this implementation at all. In particular, metacontinuations are invisible to user programs. The implementation constructs a world where user programs are as though interpreted under a tower of user-observable interpreters shown in Figure 1. The more formal relationship between Figure 1 and Figure 2 is found in [2].

---

global environment of `r`. Its definition is found in Appendix A.

3.5 Evaluator Functions as Primitives

The use of `meta-apply` is the core part of the Black system design. By restricting the reflective capability to the redefinition of functions, it achieves both the efficiency and flexibility. However, it does not provide the full power of reflection as provided by the literal execution of meta-circular interpreters [2]. If the reflective tower were implemented faithfully, there would have been another way to change the behavior of the metalevel interpreter. Rather than directly redefining the metalevel interpreter, we could go up two levels and redefine the metametalevel interpreter so that the unchanged metalevel interpreter be interpreted differently.

This type of modification is impossible in Black, since the only change the evaluator functions are aware of is the redefinition of evaluator functions through the use of `meta-apply`. Even if we replace `car` (to print out its argument, for example) at level 2, the behavior of `base-eval` at level 1 does not change. So to speak, default evaluator functions are *compiled* and treated as primitives whose internal structure is unmodifiable.

4. DS BLACK IN CPS

So far, the metalevel interpreter observed by user programs was written in CPS. We now build a variant of Black where it is written in DS. This is done by *currying* the continuation argument of evaluator functions.

4.1 Currying Evaluator Functions

How do user programs know that the metalevel interpreter is written in CPS? It is by executing the interpreter with appropriate arguments. If `base-eval` accepts `e`, `r`, and `k`, then it is written in CPS, but if `base-eval` accepts only `e` and `r`, then user programs observe it as written in DS. So, we first curry the continuation argument of evaluator functions so that they accept only `e` and `r`. As a result, the type of evaluator functions becomes

$$(Exp \times Env) \to Cont \to Mcont \to Ans \ .$$

Let us see how `eval-var` is curried. Its original definition was:

```
(define (eval-var e r k) (k (cdr (get e r))))
```

It is curried as:

```
(define (eval-var e r)
  (lambda (k) (k (cdr (get e r)))))
```

which can be rewritten as

```
(define (eval-var e r) (unit (cdr (get e r))))
```

where `unit` is defined as follows:

```
(define (unit x) (lambda (k) (k x)))
```

This function is called `unit` because it is the unit operation of the continuation monad [12]. It *returns* a result to the continuation.

`eval-var` was the easy case. To curry `eval-application`, we need to combine two calls to evaluator functions. The original definition of `eval-application` was:

```
(define (eval-application e r k)
  (meta-apply 'eval-list e r (lambda (l)
    (meta-apply 'base-apply (car l) (cdr l) r k))))
```

Assuming that `meta-apply` and other evaluator functions are curried, we curry it into:

```
(define (eval-application e r)
  (bind (meta-apply 'eval-list e r)
    (lambda (l)
      (meta-apply 'base-apply (car l) (cdr l) r))))
```

where `bind` is defined as follows:

```
(define (bind u v)
  (lambda (k) (u (lambda (x) ((v x) k)))))
```

Along this line, the evaluator functions that do not require levelshifting can be curried. The result is displayed in Figure 3. As long as continuations are simply applied to a result or passed around, they are encapsulated in `unit` and `bind` and do not appear explicitly in Figure 3. If a continuation is used in a non-standard way, however, it becomes explicit. In `meta-apply`, `cont` is explicitly received and applied because it goes beyond `shift-up` and `shift-down`.

Thanks to the use of `unit` and `bind`, the interpreter looks like written in DS. In fact, if we replace the two operations by the ones of the identity monad:

```
(define (unit x) x)
(define (bind u v) (v u))
```

then, the interpreter is actually written in DS. (We will do so in Section 5.)

4.2 Levelshifting Functions

The result of currying levelshifting functions is shown in Figure 4. The definition of `shift-up` and `shift-down` remains the same as in Section 3.3. We examine `base-apply` first. As before, it stores the current (global) environment and continuation into the metacontinuation and performs application at the level below. In contrast to the previous case, however, we have to explicitly supply a continuation used during the evaluation of the application. Since the work that has to be done after the application is simply to terminate the level and come back to the current level, `(lambda (x) (shift-up (lambda (mr mk) (mk x))))` is passed as the continuation. In the CPS case, such a continuation was supplied by user programs. In the DS case, it has to be supplied explicitly, because user programs are written in DS and do not supply continuations.

For the same reason, `eval-lambda` supplies `(lambda (x) (shift-down (k x) mr mk))` as the continuation for evaluating the body of lambda abstractions. When the evaluation finishes, the result is passed to the level below to continue the computation stored in `k`. Notice that user programs do not observe that a continuation is passed around. When a function is applied in `base-apply`, the continuation is automatically supplied by the Black interpreter. Thus, even though the internal representation receives a continuation as an extra parameter, it is hidden and not shown to user programs. In other words, user programs observe the metalevel interpreter as written in DS.

Finally, `eval-exit` stores the current environment and continuation into the metalevel environment, and returns the argument of `exit` to the metalevel. The value bound for `old-cont` becomes somewhat complicated, because it has to combine two continuations: the current one `k` and the one when `old-cont` is applied `k2`.

```
(define (base-eval e r)
  (cond ((number? e)  (unit e))
        ((boolean? e) (unit e))
        ((symbol? e)
         (meta-apply 'eval-var e r))
        ((eq? (car e) 'if)
         (meta-apply 'eval-if e r))
        ((eq? (car e) 'set!)
         (meta-apply 'eval-set! e r))
        ((eq? (car e) 'lambda)
         (meta-apply 'eval-lambda e r))
        ((eq? (car e) 'exit)
         (meta-apply 'eval-exit e r))
        ...
        (else
         (meta-apply 'eval-application e r))))
(define (eval-var e r) (unit (cdr (get e r))))
(define (eval-if e r) ; (if pred then else)
  (bind (meta-apply 'base-eval (car (cdr e)) r)
    (lambda (p)
      (if p (meta-apply
              'base-eval (car (cdr (cdr e))) r)
            (meta-apply
              'base-eval (car (cdr (cdr (cdr e))))
                         r)))))
(define (eval-set! e r) ; (set! var exp)
  (bind (meta-apply
          'base-eval (car (cdr (cdr e))) r)
    (lambda (data)
      (set-value! (car (cdr e)) data r)
      (unit (car (cdr e))))))
(define (eval-application e r) ; (f a b c ...)
  (bind (meta-apply 'eval-list e r) (lambda (l)
    (meta-apply 'base-apply (car l) (cdr l) r))))
(define (eval-list e r)
  (if (null? e) (unit '())
    (bind (meta-apply 'base-eval (car e) r)
      (lambda (u)
        (bind (meta-apply 'eval-list (cdr e) r)
          (lambda (v) (unit (cons u v))))))))
(define (meta-apply proc-name . operand)
  (lambda (k)
    (shift-up (lambda (mr mk)
      (shift-down
        ((scheme-apply (cdr (get proc-name mr))
                       operand)
         k)
        mr mk)))))
```

Figure 3: Implementation of DS Black in CPS (nonlevelshifting functions and `meta-apply`)

```
(define (eval-lambda e r) ; (lambda params body)
  (let ((body (car (cdr (cdr e))))
        (params (car (cdr e))))
    (unit (lambda operand (lambda (k)
            (shift-up (lambda (mr mk)
              ((meta-apply 'base-eval body
                        (extend r params operand))
               (lambda (x)
                 (shift-down (k x) mr mk))))))))))
(define (eval-exit e r) (lambda (k) ; (exit exp)
  (shift-up (lambda (mr mk)
    (set-value! 'old-env env mr)
    (set-value! 'old-cont
                (lambda (x) (lambda (k2)
                  (shift-up (lambda (mr2 mk2)
                    (shift-down (k x)
                      mr2
                      (lambda (y)
                        (shift-down (k2 y)
                          mr2 mk2)))))))
                mr)
    (mk (car (cdr e)))))))
(define (base-apply operator operand r)
  (lambda (k)
    (shift-down ((scheme-apply operator operand)
                 (lambda (x)
                   (shift-up (lambda (mr mk)
                     (mk x)))))
                (get-global-env env) k)))
```

Figure 4: Implementation of DS Black in CPS (levelshifting functions)

4.3 Example Execution

Let us see some example interaction with the metalevel interpreter written in DS.

```
> (black)
0-0: start
0-1> (car (cons 1 2))
0-1: 1
0-2> (exit 0)
New level loaded.
1-0: 0
1-1> base-eval
1-1: #[procedure base-eval]
1-2>
```

So far, it behaves exactly the same as before. However, the procedure `base-eval` is now written in DS. It accepts only two arguments.

```
1-2> (base-eval '(+ 1 2) init-env)
1-2: 3
1-3>
```

We can change it similarly as before, but in DS now. This time, let us print not only the expression to be evaluated but also the result.

```
1-3> (define old-eval base-eval)
1-3: old-eval
1-4> (set! base-eval
           (lambda (e r)
             (write '->:) (write e) (newline)
             (let ((x (old-eval e r)))
               (write '<-:) (write x) (newline)
               x)))
1-4: base-eval
1-5>
```

We used a `let` construct here, which can be easily added to the system. Alternatively, we could go up to level 2 and redefine the metametalevel interpreter so that it understands `let` constructs.

After going back to level 0, the change of `base-eval` becomes effective.

```
1-5> (old-cont 1)
0-2: 1
0-3> (car (cons 1 2))
->:(car (cons 1 2))
->:car
<-:#[procedure car]
->:(cons 1 2)
->:cons
<-:#[procedure cons]
->:1
<-:1
->:2
<-:2
<-:(1 . 2)
<-:1
0-3: 1
0-4>
```

The same thing can be done with the CPS metalevel interpreter, if we write it in CPS. By changing the metalevel interpreter into DS, it becomes much simpler to program such extensions.

4.4 Limitation

Although technically correct, the DS Black exhibits some troublesome behavior. If we exit after the above transcript, we obtain the following:

```
0-4> (exit 0)
->:(exit 0)
->:0
<-:0
<-:0
0-4: 0
0-5>
```

We cannot exit to level 1 but come back to level 0 again. It happens by the following reason: when `(old-eval e r)` in the redefined `base-eval` is executed, a new level is launched and it is executed at the level below; during the execution, `(exit 0)` is input; the current continuation (of the launched level) is discarded and 0 is returned to the original level; it becomes the value of `(old-eval e r)` (because it is the only expression executed at the launched level) and proceeds as though it returned a result normally. The problem here is that the printing of traces is not included in the captured continuation of the launched level. Although we intend that printing traces is a level 1 activity where `old-eval` is running, we have no way to declare it.

It is illustrating to see what happens if we do the same thing in CPS Black. In CPS Black, we usually redefine `base-eval` as follows:

```
(lambda (e r k)
  (write '->:) (write e) (newline)
  (old-eval e r (lambda (x)
    (write '<-:) (write e) (newline)
    x)))
```

The point here is that the printing of the result is in the continuation argument of `old-eval`. With this `base-eval`, we can exit level 0, because the continuation which prints the result is discarded when `exit` is executed. However, if we redefine `base-eval` as follows:

```
(lambda (e r k)
  (write '->:) (write e) (newline)
  (let ((x (old-eval e r (lambda (y) y))))
    (write '<-:) (write x) (newline)
    x))
```

then, the same situation as DS Black occurs. Since we use `(lambda (y) y)` as the continuation of `old-eval` and DS application to print the result of evaluation, the continuation discarded at the evaluation of `(exit 0)` will be only `(lambda (y) y)` and not include printing of the result.

In CPS Black, we have the first option. In DS Black, we have only the second option so far. If the program is written in the tail-reflective manner [4], *i.e.*, in a way levelshifting occurs only at the tail position, we can avoid the problem by treating the tail continuation (which performs only going up a level) specially. However, more general treatment requires full inspection of continuations. It exhibits a miss match between the current strategy of shifting levels and our understanding of levels. Further work is required here.

5. DS BLACK IN DS

The DS Black shown in the previous section itself has been written in CPS. In this section, we write the same system in DS.

5.1 New shift-up and -down

The main tool we use to make the whole system written in DS is `shift` and `reset`. In the DS Black written in CPS, continuations are passed around using an extra parameter. When necessary, evaluator functions are η-expanded to explicitly receive continuations. We achieve the same effect by storing continuations in the stack of the underlying Scheme system, and capturing them using `shift`. We use the following two functions:

```
(define (shift-up code)
  (lambda (Mcont)
    (let ((mr (car (head Mcont)))
          (mk (car (cdr (head Mcont))))
          (meta-Mcont (tail Mcont)))
      ((reset (code mr mk)) meta-Mcont))))
(define-macro (shift-down code r k)
  `(lambda (Mcont)
     ((reset ,code)
      (cons-stream (list ,r ,k) Mcont))))
```

Roughly speaking, we encode a level as a `reset` applied to a metacontinuation as in `((reset code) Mcont)`. Since the `code` here is typically a read-eval-print loop, it does not usually return any value. Thus, without shifting up or down, `Mcont` is not used, and all the computation proceed within `reset`.

To go up a level, we execute `(shift k (shift-up code))`. We first grab the current continuation up to the enclosing `reset`. Since `(lambda (Mcont) ...)` returned by `shift-up` is a value, it then escapes out of the enclosing `reset`, and receives the metacontinuation. Finally, `code` supplied with `mr` and `mk` is executed under `meta-Mcont`, *i.e.*, at the metalevel. At this time, the execution is again enclosed by `reset` for the possible levelshifting during the execution of `code`. Remember that in the previous CPS version, we wrote it as `(lambda (k) (shift-up code))`. We now use `shift` to capture `k`.

To go down a level, we write `(shift-down code r k)`.[3] We use `shift-down` only at the top level of the enclosing `reset`, so that `(lambda (Mcont) ...)` is returned as a value of `reset`. It then receives a metacontinuation, and execute `code` at the level below.

5.2 Implementation in DS

We are now ready to write the Black reflective interpreter in DS. See Figure 5. Because the interpreter is written using `unit` and `bind`, the textual definition of the non-levelshifting evaluator functions is exactly the same as in Figure 3. But here, the definition of `unit` and `bind` is different:

```
(define (unit x) x)
(define (bind u v) (v u))
```

They are instantiated to the unit and bind operations for the identity monad. Namely, the interpreter is actually written in DS.

The definition of levelshifting functions are also similar to the previous case. There are only two differences. One

```
(define (base-eval e r)
  the same as in Figure 3)
(define (eval-var e r)
  the same as in Figure 3)
(define (eval-if e r) ; (if pred then else)
  the same as in Figure 3)
(define (eval-set! e r) ; (set! var exp)
  the same as in Figure 3)
(define (eval-lambda e r) ; (lambda params body)
  (let ((body (cdr (cdr e)))
        (params (car (cdr e))))
    (unit
      (lambda operand (shift k
        (shift-up (lambda (mr mk)
          ((lambda (x) (shift-down (k x) mr mk))
           (meta-apply 'eval-begin body
                (extend r params operand)))))))))))
(define (eval-exit e r) ; (exit exp)
  (shift k (shift-up (lambda (mr mk)
    (set-value! 'old-env r mr)
    (set-value! 'old-cont
      (lambda (x) (shift k2
        (shift-up2 (lambda (mr2 mk2)
          (shift-down (k x) mr2
            (lambda (y)
              (shift-down (k2 y) mr2 mk2)))))))
      mr)
    (mk (car (cdr e)))))))
(define (eval-application e r) ; (f a b c ...)
  the same as in Figure 3)
(define (eval-list e r)
  the same as in Figure 3)
(define (base-apply operator operand r)
  (shift k (shift-down
    ((lambda (x)
       (shift-up (lambda (mr mk) (mk x))))
     (scheme-apply operator operand))
    (get-global-env r) k)))
(define (meta-apply proc-name . operand)
  (shift k
    (shift-up (lambda (mr mk)
      (shift-down
        (k (scheme-apply (cdr (get proc-name mr))
                         operand))
        mr mk)))))
```

Figure 5: Implementation of DS Black in DS

---

[3] Since `code` needs to be executed in the scope of `reset`, `shift-down` is defined as a macro rather than a function. We assume that the macro system is hygienic [5] so that the name `Mcont` appearing in the definition is distinct from any other variables in the program.

is that they use `shift` to grab the current continuation rather than receiving as an argument. To be more concrete, `(lambda (k) (shift-up ...))` in Figure 4 becomes `(shift k (shift-up ...))` in Figure 5. The other is that the continuations passed to evaluator functions are directly applied rather than passed as an argument. Thus, `((...) cont)` becomes `(cont (...))`.

5.3 Benefits of Writing Black in DS

In which style, CPS or DS, the Black system is implemented is irrelevant, since the user-observable metalevel interpreter is identical. However, there is an important benefit of writing Black system in DS. It becomes much simpler to make small functions public. If Black is written in CPS, all the redefinable functions must be written in CPS. It means that if we want functions like `get` and `extend` to be redefinable, then we have to rewrite them in CPS. If Black is written in DS, on the other hand, we can make them redefinable simply by inserting `meta-apply` whenever we call them.

As another point, compilation (or partial evaluation) of redefined functions would become easier. Since redefined evaluator functions are interpreted by the interpreter one level above, they incur significant overhead. Compiling them into directly executed code can remove such overhead. If Black is written in CPS using a metacontinuation, compilation has to take care of it, making compilation complicated. By separating levels as a delimited context, a unit of compilation becomes clear and compilation would become easier. We now have to handle `shift` and `reset` during compilation, but a theory for control operators in compilation (partial evaluation) is being developed [1].

6. DISCARDING METACONTINUATIONS

When the control operators `shift` and `reset` were introduced, Danvy and Filinski pointed out the similarity between these operators and levelshifting in the reflective tower [3]: they both use a metacontinuation. It suggests that we might be able to encode levelshifting in the reflective tower by nesting structure of `reset` without using metacontinuations. In this section, we investigate and clarify their similarity as well as their difference, which enables us to encode `shift-up` and `shift-down` in terms of `shift` and `reset` without using metacontinuations. As was the case in the previous section, the intuition is that we represent a level by a delimited context. `Reset` is used to create a new level and `shift` is used to go up a level. Unlike the previous section, however, we do not use metacontinuations, but use the nesting of `reset` directly to represent levels.

The difference between `shift/reset` and `shift-up/down` can be summarized in the following two points: (1) `shift` captures only the continuation while `shift-up` captures both the environment and continuation, (2) after capturing a continuation, `shift` does not go out of the enclosing `reset` but stays in its scope, whereas `shift-up` goes out of the current level. The second point is crucial because it means that even though we use `shift` in the body of another `shift`, we cannot go out of the enclosing `reset`. Thus, if we used `shift` directly to represent `shift-up`, then we could not go up two levels by nesting `shift-up`. `Shift/reset` is statically scoped, but `shift-up/down` is not.

To bridge the gap, we use the following encodings:

```
(define (base-eval e r)
  the same as in Figure 3)
(define (eval-var e r)
  the same as in Figure 3)
(define (eval-if e r) ; (if pred then else)
  the same as in Figure 3)
(define (eval-set! e r) ; (set! var exp)
  the same as in Figure 3)
(define (eval-lambda e r) ; (lambda params body)
  (let ((body (cdr (cdr exp)))
        (params (car (cdr exp))))
    (unit
      (lambda operand
        (shift-up (lambda (mr k)
          ((lambda (x) (shift-down (k x) mr))
           (meta-apply 'base-eval body
                 (extend r params operand)))))))))
(define (eval-exit e r) ; (exit exp)
  (shift-up (lambda (mr k)
    (set-value! 'old-env r mr)
    (set-value! 'old-cont
      (lambda (x)
        (shift-up (lambda (mr2 k2)
                    (k2 (shift-down (k x) mr2)))))
      mr)
    (car (cdr e)))))
(define (eval-application e r) ; (f a b c ...)
  the same as in Figure 3)
(define (eval-list e r)
  the same as in Figure 3)
(define (base-apply operator operand r)
  (shift-down ((lambda (x)
                 (shift-up (lambda (mr k) (k x))))
               (scheme-apply operator operand))
    (get-global-env r)))
(define (meta-apply proc-name . operand)
  (shift-up (lambda (mr k)
    (shift-down
      (k (scheme-apply (cdr (get proc-name mr))
                       operand))
      mr))))
```

Figure 6: Implementation of DS Black in DS without metacontinuation

```
(define (shift-up code)
  (shift k (lambda (mr) (code mr k))))
(define-macro (shift-down code mr)
  `((reset ,code) ,mr))
```

We break the static scoping of `shift/reset` by inserting a lambda abstraction that take the metalevel environment. In the previous section, we used the same technique to pass a metacontinuation rather than the metalevel environment.

Here, a level is represented as a `reset` applied to the metalevel environment `((reset code) mr)` as shown in the `shift-down`. Since the `code` here is typically a read-eval-print loop, it does not usually return any value. To go up a level, we use `shift-up`. It first grabs the continuation up to the enclosing `reset`, and then returns a lambda abstraction that accepts a metalevel environment: `(lambda (mr) ...)`. Because it is a value, it escapes the enclosing `reset`, thus terminating the current level. The lambda abstraction then receives the metalevel environment applied to `reset`, and the rest of the computation occurs at the metalevel.

Before the argument `code` of `shift-up` is evaluated at the metalevel, it receives the metalevel environment and the captured (current-level) continuation. Remember that in the previous case, it received the metalevel environment and the *metalevel* continuation. In the new encodings, the metalevel continuation is the context in which `shift-up` is executed, and is implicit here. The similar comment applies to `shift-down`. Since all the continuations are implicit, `shift-down` does not receive any continuation as its argument.

Using these encodings, the DS Black written in DS without metacontinuations is shown in Figure 6. As expected, non-levelshifting functions are exactly the same as before. The definition of levelshifting functions is also similar. It goes parallel to Figure 5. Since the metalevel continuation `mk` is implicit, Figure 6 manipulates only `mr` and `k`, where as Figure 5 manipulates `mk`, too.

7. RELATED WORK

A Lisp-based reflective language was first presented by Smith in the language called 3-LISP [11]. It is followed by two reflective languages: Brown [6, 13] and Blond [4], which are both implemented in CPS using a metacontinuation. Jefferson and Friedman implemented a simple reflective interpreter [8] by writing a meta-circular interpreter and runs it on top of itself. Although it suffers from severe efficiency problem, the technique used there is not restricted to CPS but can be applied to DS, too.

This work is the continuation of the previous work [2] where the main issue was to establish how to implement a reflective system in which all the evaluator functions are redefinable. The system was written in CPS using a metacontinuation. The present work concentrates on how to write a similar system in which both the user-observable interpreter and the actual implementation are written in DS.

The similarity between the control operators `shift/reset` and the reflective tower was pointed out by Danvy and Filinski [3]. This paper pursued the similarity to encode `shift-up/down` using `shift/reset`. This enabled us to implement a reflective language in direct style without using a metacontinuation.

8. CONCLUSION

This paper has described a reflective language in which the user-observable metalevel interpreter is written in DS rather than in CPS. It enables user programs to modify the language semantics in DS. Three versions of such systems are shown. The first one is written in CPS with metacontinuations, the second one in DS with metacontinuations, and the third one in DS without metacontinuations.

9. ACKNOWLEDGEMENTS

I received comments and encouragements from Olivier Danvy.

10. REFERENCES

[1] Asai, K. “Online Partial Evaluation for Shift and Reset,” *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM ’02)*, pp. 19–30 (January 2002).

[2] Asai, K., S. Matsuoka, and A. Yonezawa “Duplication and Partial Evaluation — For a Better Understanding of Reflective Languages —,” *Lisp and Symbolic Computation*, Vol. 9, Nos. 2/3, pp. 203–241, Kluwer Academic Publishers (May/June 1996).

[3] Danvy, O., and A. Filinski “Abstracting Control,” *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160 (June 1990).

[4] Danvy, O., and K. Malmkjær “Intensions and Extensions in a Reflective Tower,” *Conference Record of the 1988 ACM Symposium on Lisp and Functional Programming*, pp. 327–341 (July 1988).

[5] Dybvig, R. K., R. Hieb, and C. Bruggeman “Syntactic Abstraction in Scheme,” *Lisp and Symbolic Computation*, Vol. 5, No. 4, pp. 295–326, Kluwer Academic Publishers (December 1992).

[6] Friedman, D. P., and M. Wand “Reification: Reflection without Metaphysics,” *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 348–355 (August 1984).

[7] Ichisugi, Y., S. Matsuoka, and A. Yonezawa “RbCl: A Reflective Object-Oriented Concurrent Language without a Run-time Kernel,” *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, pp. 24–35 (November 1992).

[8] Jefferson, S., and D. P. Friedman “A Simple Reflective Interpreter,” *Lisp and Symbolic Computation*, Vol. 9, Nos. 2/3, pp. 181–202, Kluwer Academic Publishers (May/June 1996).

[9] Kelsey, R., W. Clinger, and J. Rees (editors) “Revised^5 Report on the Algorithmic Language Scheme,” *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, pp. 7–105, Kluwer Academic Publishers (August 1998).

[10] Lawall, J. L., and O. Danvy “Continuation-Based Partial Evaluation,” *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pp. 227–238 (June 1994).

[11] Smith, B. C. “Reflection and Semantics in Lisp,” *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, pp. 23–35 (January 1984).

[12] Wadler, P. “The Essence of Functional Programming,” *Conference Record of the 19th Annual*

ACM Symposium on Principles of Programming Languages, pp. 1–14 (January 1992).

[13] Wand, M., and D. P. Friedman "The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower," *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming*, pp. 298–307 (August 1986).

APPENDIX

A. ENVIRONMENT FUNCTIONS

The appendix shows the implementation of environments used in the paper. An environment is a list of frames, and a frame is an association list.

```
(define empty-env '(()))
(define (make-pairs params args)
  (cond ((null? params) '())
        ((symbol? params) ; for dotted parameter
         (list (cons params args)))
        (else
         (cons (cons (car params) (car args))
               (make-pairs (cdr params)
                           (cdr args))))))
(define (extend env params args)
  (cons (make-pairs params args) env))
(define (get var env)
  (if (null? env)
      '() ; unbound variable
      (let ((pair (assq var (car env))))
        (if (pair? pair)
            pair
            (get var (cdr env))))))
(define (set-value! var value env)
  (set! (get var env) value))
(define (get-global-env env)
  (if (null? (cdr env))
    env
    (get-global-env (cdr env))))
```