Reflection in Direct Style

Kenichi Asai
Ochanomizu University
asai@is.ocha.ac.jp

Abstract

A reflective language enables us to access, inspect, and/or modify the language semantics from within the same language framework. Although the degree of semantics exposure differs from one language to another, the most powerful approach, referred to as the behavioral reflection, exposes the entire language semantics (or the language interpreter) that defines behavior of user programs for user inspection/modification. In this paper, we deal with the behavioral reflection in the context of a functional language Scheme. In particular, we show how to construct a reflective interpreter where user programs are interpreted by the tower of metacircular interpreters and have the ability to change any parts of the interpreters during execution. Its distinctive feature compared to the previous work is that the metalevel interpreters observed by users are written in direct style. Based on the past attempt of the present author, the current work solves the level-shifting anomaly by defunctionalizing and inspecting the top of the continuation frames. The resulting system enables us to freely go up and down the levels and access/modify the direct-style metalevel interpreter. This is in contrast to the previous system where metalevel interpreters were written in continuation-passing style (CPS) and only CPS functions could be exposed to users for modification.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures; D.3.4 [*Programming Languages*]: Processors—Interpreters

General Terms Languages

Keywords Reflection, metacircular interpreter, metacontinuation, continuation-passing style (CPS), direct style, partial evaluation

1. Introduction

A reflective language enables us to access, inspect, and/or modify the language semantics from within the same language framework. Originally, reflection was proposed by Smith in his pioneering work on 3-LISP [15], where user programs were executed by an infinite number of metacircular interpreters (called a reflective tower) and had access to the expression, environment, and continuation of the current computation. Since these pieces of information determine the complete state of computation, user programs effectively have control over how the state of computation is manipulated, in other words, the language semantics itself. This kind of reflection is called behavioral reflection.

GPCE'11, October 22–23, 2011, Portland, Oregon, USA.

The theoretical idea of 3-LISP was followed by two reflective languages, Brown [6, 17] and Blond [5], which explained switching of levels using metacontinuations and improved on the efficient execution of programs under a tower of interpreters. The ability to change the metalevel interpreter, as opposed to simply having access to the state of computation, was added to these languages by the present author in the reflective language Black [1, 2]. In this language, the metalevel interpreter (or the operational semantics of the language) is open to user programs as a collection of standard functions and is subject to change at runtime. User programs have not only access to the state of computation but also ability to change the operational semantics directly.

The idea of reflection affected the design of programming languages in various ways. In the object-oriented language CLOS (Common Lisp Object System), metaobject protocol [11] was used to grant user programs access to metaobjects that define the semantics of baselevel objects as a kind of metacircular interpreter. Similar idea was applied in the concurrent object-oriented language ABCL/R3 [13] to tune and optimize the behavior of concurrent objects. The idea of reflection was further developed into Aspect-Oriented Programming [12], where various kinds of semantic aspects (among many cross-cutting concerns) are modularized and made public for user control.

Although reflective capabilities are strong and useful, the reflective mechanisms provided in most languages are restricted, because it is difficult to efficiently execute reflective programs, in particular the ones that use behavioral reflection. For example, Java allows to access various information via reflection, but does not allow radical changes to the language semantics. On the other hand, it is difficult to predict all the reflective capabilities that could be useful beforehand. Thus, whenever we need new reflective capabilities, we need to modify the underlying language implementation.

To remedy this situation, the original approach to the behavioral reflection using metacircular interpreters is gaining interest. Most notably, Verwaest et al. [16] recently proposed a reflective system which adopts Smalltalk-like object model with a tower of first-class interpreters. Because the full language semantics is exposed to user programs as a first-class interpreter, any modification to the language semantics is possible.

To back up such work and provide its foundation, we deal with the behavioral reflection in the context of a functional language Scheme. In particular, we consider how to build a reflective system in which the metalevel interpreter is written without any restriction. In our previous work [2], we have already proposed a general method to build a reflective system, but the method crucially depended on the fact that the metalevel interpreter was written in continuation-passing style (CPS). In this paper, we remove this restriction and allow for the metalevel interpreter to be written in the ordinary direct style. In our past attempt [1], we have already constructed such a reflective system, but it was not quite right because it suffered from the level-shifting anomaly. This paper clarifies why the anomaly occurs and shows one possible method to avoid it. As

a result, the current system becomes the first such system without the anomaly.

The system is easier to use than before, since we do not have to program in CPS any more. We shall demonstrate a number of example executions in the paper. Theoretically, this paper clarifies how to build a reflective system where the metalevel interpreters are written in direct style. More practically, it gives a foundation upon which advanced features such as runtime specialization can be built for efficient execution of reflective programs.

2. Preliminaries: `shift` and `reset`

We will use the delimited control operators [4], `shift` and `reset`, in this paper. The `shift` operator captures and clears the current continuation up to the enclosing `reset` operator and binds it to its first argument before executing its second argument. For example, in the following expression

```
(+ 1 (reset (* 2 (shift k (k (k 3))))))
```

the captured continuation bound to `k` is `(* 2 □)`. It is applied twice in the body of `shift`, so the `shift` expression evaluates to 12. Since the captured continuation is cleared, it becomes the value of `reset` and the final answer becomes 13.

The `shift` operator is like `call-with-current-continuation`, but is different in that the captured continuation is delimited and thus composable, and that the captured continuation is cleared. We will use the delimited context as a representation of a level, and use `shift` to capture the continuation of the current level.

3. The reflective language Black

The reflective language we use in this paper is a Scheme-based language called Black [1, 2]. It is a standard Scheme interpreter with two new reflective constructs, `EM` and `exit`. The former stands for *Execute at Metalevel* and evaluates its argument at the metalevel, namely, at the level of the interpreter the current expression is evaluated. The latter is used to finish the current level and go up to the metalevel. The interaction between `exit` and the reflective framework in this paper will become one of the interesting topics of this paper.

To see how `EM` works, we demonstrate some example execution of Black.

```
> (black)
0-0: start
0-1> (* 2 (+ 1 4))
0-1: 10
0-2>
```

It is basically a standard Scheme interpreter. The first number `0` in the prompt indicates the current level. The second number indicates the number of iterations in the current level. To evaluate an expression at the metalevel, we use `EM`.

```
0-2> (EM (* 2 (+ 1 4)))
0-2: 10
0-3>
```

When `EM` is executed, its argument is sent as is to the metalevel for execution. At the metalevel, the expression `(* 2 (+ 1 4))` is evaluated to 10 and the result is sent back to the current level.

We can observe the metalevel interpreter using `EM`.

```
0-3> (EM base-eval)
0-3: #<procedure #2 base-eval>
0-4>
```

The main function of the metalevel interpreter is called `base-eval`. We can even modify the definition of `base-eval` executing the user programs. To do so, however, we need to know how the metalevel interpreter is written. Figure 1 shows the main parts of the metalevel interpreter. User programs are *supposed* to be executed by this interpreter. In other words, this is the interpreter observed by the user programs whenever they access it using `EM`. However, the actual Black interpreter is different from this user-observable interpreter. The Black interpreter executes user programs as though they are interpreted by the user-observable interpreter. It defines the API to access the metalevel, or the Metaobject Protocol, so to speak.

The user-observable interpreter is a standard eval/apply-style interpreter written in monadic style. Given an expression `e` and an environment `r`, the main function `base-eval` dispatches over `e`, and executes one of `eval-*` functions. We call these functions that comprise an interpreter *evaluator functions*. Lambda closures are represented as a tagged list, where the address (`eq`-ness) of `lambda-tag` is used to distinguish closures from other data.

The interpreter is standard except for three points. First, it includes `eval-EM` to interpret the `EM` construct. Its definition is (almost) metacircular: a special construct `primitive-EM` is used to interpret `EM`. The former is a primitive version of `EM` where the argument is evaluated before it is sent to the metalevel. The important point here is that the behavior of `eval-EM` cannot be described without the help of such a special construct. Since `EM` escapes the current level, its precise behavior can be described only in terms of the reflective tower. Thus, the user-observable interpreter can only include the definition of `eval-EM` that *magically* evaluates its argument at the metalevel.

Secondly, the definition of `my-error` in the user-observable interpreter is disappointing. It is supposed to print an error message and abort the current execution, but it is defined as returning 0 normally. Thus, if an error occurs (*e.g.*, by applying a boolean to an argument; see the `else` branch of `base-apply`), the result becomes 0 unconditionally and the execution continues which could result in further errors.

```
0-4> ((#t 3) 4)
(Not a function: #t)
(Not a function: 0)
0-4: 0
0-5>
```

In this example, `(#t 3)` results in an error and the corresponding error message is printed, but since its result becomes 0, it incurs further error at the outer application `(0 4)`. Only the first error message is valid and the rest of the computation is completely bogus. For now, the user-observable interpreter is defined like this, because aborting the current execution requires special treatment, such as introduction of an error value or first-class continuation constructs. We will come back to this problem first in Section 4.4 and then more seriously in Section 5.

Finally, the user-observable interpreter is written in monadic style. It might first appear to contradict what this paper claims to achieve, because CPS is an instance of monadic style and the interpreter can be thought of as written in CPS rather than direct style. This is not the case, since the two monadic operators, `unit` and `bind`, in the user-observable interpreter do not have any special status. They are just other functions. The interpreter is written in direct style, because nested function calls are used. For example, in `eval-if`, the result of applying `base-eval` is passed to `bind`. We use monadic-style interpreter in this paper because we can change the language semantics in an interesting way by modifying `unit` and `bind`. Alternatively, we could have inlined the definition of `unit` and `bind` and started from a direct-style interpreter: all the rest of the story equally holds (except that we can no longer replace `unit` and `bind` because they would not exist any more).

```
(define (unit x) x)
(define (bind u v) (v u))
(define (base-eval e r)
  (cond ((number? e)           (unit e))
        ((boolean? e)          (unit e))
        ((symbol? e)           (eval-var e r))
        ((eq? (car e) 'if)     (eval-if e r))
        ((eq? (car e) 'set!)   (eval-set! e r))
        ((eq? (car e) 'lambda) (eval-lambda e r))
        ((eq? (car e) 'EM)     (eval-EM e r))
        ((eq? (car e) 'exit)   (eval-exit e r))
        ...
        (else (eval-application e r))))
(define (eval-var e r) (cdr (get e r)))
(define (eval-if e r) ; (if pred then else)
  (bind (base-eval (car (cdr e)) r)
    (lambda (pred)
      (if pred
        (base-eval (car (cdr (cdr e))) r)
        (base-eval (car (cdr (cdr (cdr e)))) r)))))
(define (eval-set! e r) ; (set! var body)
  (let ((var (car (cdr e)))
        (body (car (cdr (cdr e)))))
    (bind (base-eval body r)
      (lambda (data)
        (set-value! var data r)
        (unit var)))))
(define lambda-tag (cons 'lambda 'tag))
(define (eval-lambda e r) ; (lambda params body)
  (let ((params (car (cdr e)))
        (body (car (cdr (cdr e)))))
    (unit (list lambda-tag params body r))))
(define (eval-EM e r) ; (EM exp)
  (primitive-EM (car (cdr e))))
(define (eval-exit e r) ; (exit exp)
  (bind (base-eval (car (cdr exp)) r)
    (lambda (v) (my-error v r))))
(define (eval-application e r) ; (f a b c ...)
  (bind (eval-list e r)
    (lambda (l) (base-apply (car l) (cdr l) r))))
(define (eval-list e r)
  (if (null? e)
    (unit '())
    (bind (base-eval (car e) r)
      (lambda (val1)
        (bind (eval-list (cdr e) r)
          (lambda (val2)
            (unit (cons val1 val2))))))))
(define (base-apply op args r)
  (cond ((procedure? op)
         (unit (apply op args)))
        ((and (pair? op)
              (eq? (car op) lambda-tag))
         (let ((params        (car (cdr op)))
               (body     (car (cdr (cdr op))))
               (env (car (cdr (cdr (cdr op))))))
           (base-eval body
             (extend env params args))))
        (else
         (my-error
           (list 'Not 'a 'function: op) r))))
(define (my-error e r)
  (write e) (newline) (unit 0))
```

Figure 1. The metalevel interpreter observed by users

Now that we know how the metalevel interpreter is written, we are ready to modify it. First, we save the original `base-eval` before modifying it.[1]

```
0-5> (EM (define old-eval base-eval))
0-5: old-eval
0-6>
```

The standard example is to change `base-eval` so that it prints the expression to be evaluated before actually evaluating it:

```
0-6> (EM (set! base-eval
           (lambda (e r)
             (write e) (newline) (old-eval e r))))
0-6: base-eval
0-7> (* 2 (+ 1 4))
(* 2 (+ 1 4))
*
2
(+ 1 4)
+
1
4
0-7: 10
0-8>
```

After replacing `base-eval`, the user program is interpreted by the modified interpreter. More examples will be shown in the subsequent sections.

4. How to construct Black

Given the user-observable interpreter in a metacircular way, it is not so difficult to construct a reflective tower by actually executing the interpreter on top of itself and handling reflective constructs specially. In fact, Jefferson and Friedman [8] implemented a simple reflective interpreter in this way. However, it suffers from at least two problems. First, the number of interpreters in the reflective tower has to be fixed beforehand. Secondly, the execution of user programs is extremely slow, due to the interpretive overhead. To make the metalevel interpreter modifiable, we need to interpret the metalevel interpreter using another interpreter. Thus, the user programs are interpreted by two interpreters. Usually, one more level of interpretation leads to order of magnitude slow down. This slow down is unavoidable, even if we do not use any reflective capabilities.

4.1 Interpreted vs. compiled code

To avoid the overhead of double interpretation, triple interpretation, etc., the Black system introduces distinction between interpreted code and compiled code. Interpreted code is sensitive to the redefinition of the metalevel interpreter because it is interpreted by it: after the metalevel interpreter is modified, the code will be executed under the modified interpreter. On the other hand, compiled code is insensitive to the redefinition of the metalevel interpreter because it is already compiled and is directly executed in machine code. Even if the metalevel interpreter is modified, the compiled code behaves the same as before.

By distinguishing two kinds of code, it becomes possible to achieve both the redefinability of metalevel interpreters and efficient execution. When Black is launched, all the functions in the metalevel interpreter are compiled code. Thus, the user programs are efficiently interpreted without requiring double interpretation.

[1] The metalevel interpreter in Figure 1 does not have `define` special form, but it is easy to support it. We will also use other standard special forms in the paper.

When user programs use reflection and modify a part of the interpreter, that modified part is replaced with an interpreted code. The point here is that until the metalevel interpreter is modified, user programs are interpreted efficiently, and even after it is modified, since only modified part is replaced with an inefficient interpreted code, most parts of the interpreter remain efficient compiled code.

With the introduction of compiled code, we no more have to fix the height of the reflective tower beforehand. When user programs use reflective capabilities and execute code at the metalevel, we lazily create the metametalevel interpreter to interpret the user code. If user goes up further (by a nested use of `EM`), more interpreters are created on demand.

Note that this design of a reflective tower does *not* keep all the power of reflection. Even if we go up two levels and modify the metametalevel interpreter, it does not affect the behavior of the metalevel interpreter because it is not interpreted. The introduction of compiled code means that we keep only the ability to *replace* the metalevel interpreter and abandon the ability to *modify inner workings* of the interpreter (without replacing it) in favor of efficient execution. More discussion on this point as well as the general method how to construct (CPS-based) reflective systems is found in [2].

4.2 Hook

The ability to replace the metalevel interpreter of Black is achieved by inserting *hooks* whenever a function is called. For example, `eval-application` makes three function calls, `bind`, `eval-list`, and `base-apply` (if we ignore the two calls to primitives):

```
(define (eval-application e r) ; (f a b c ...)
  (bind (eval-list e r)
    (lambda (l)
      (base-apply (car l) (cdr l) r))))
```

At each function call, we insert a call to `meta-apply` as follows:

```
(define (eval-application e r) ; (f a b c ...)
  (meta-apply 'bind
    (meta-apply 'eval-list e r)
    (lambda (l)
      (meta-apply 'base-apply (car l) (cdr l) r))))
```

The role of `meta-apply` is to check whether the called function is redefined or not and call an appropriate function. If it is not, `meta-apply` calls the default compiled code directly. If it is redefined to a user-defined (interpreted) function, on the other hand, `meta-apply` calls the interpreter one level above to interpret the redefined interpreted function. This way, `meta-apply` bridges a gap between compiled code and interpreted code. The exact definition of `meta-apply` is shown in the next section.

Where to insert `meta-apply` is arbitrary. If we insert it, the function call becomes sensitive to redefinition. If we do not insert it, the function call becomes insensitive to redefinition. In the current Black implementation, we hook all the evaluator functions and avoid hooking primitive functions and small environment manipulating functions.

4.3 Shifting levels

In Black, non-level-shifting functions are implemented simply by inserting `meta-apply` to appropriate places. The implementation of level-shifting functions, on the other hand, involves manipulation of levels. Before showing how level-shifting functions are implemented in Black, we first explain how levels are handled in Black.

To represent an infinite tower of interpreters, we use a *metacontinuation* stored in a global variable `Mcont`. A metacontinuation consists of a lazy stream containing a pair of an environment and a continuation for each level, starting from the current level. We use `cons-stream` to create a lazy stream (whose tail part is delayed) and `head` and `tail` to extract head and tail parts of a stream (where the tail part is forced when extracted). Using a metacontinuation, shift up and down can be implemented as follows:

```
(define (shift-up code)
  (let ((meta-env (car (head Mcont)))
        (meta-cont (car (cdr (head Mcont))))
        (meta-Mcont (tail Mcont)))
    (set! Mcont meta-Mcont)
    (code meta-env meta-cont)))
(define-macro (shift-down code env cont)
  `(begin
     (set! Mcont (cons (list ,env ,cont) Mcont))
     ,code))
```

Shifting up a level is implemented as popping the environment and continuation at the top of `Mcont` and executing `code` at the metalevel with the popped values. Shifting down a level is implemented as pushing the environment and continuation to `Mcont` before executing `code`.[2] Note that `shift-down` is implemented as a macro, since the argument `code` has to be executed under the new `Mcont`.

To use these two functions, we somehow need to capture the current continuation to save it into the metacontinuation and install a continuation restored from the metacontinuation. The latter is easy: we just apply the continuation to a result. To accommodate the former, we use a control operator `shift` and maintain an invariant that the current level is delimited by `reset`.

Now, level-shifting functions are implemented as follows. See Figure 2. When `meta-apply` is called, it first captures and clears the current continuation using `shift`. It then shifts up one level and checks whether the called function is redefined or not by consulting the metalevel environment `mr`. If the called function is bound to a procedure, it means that it is a (default) compiled function. In this case, it is applied to the arguments in the original level under the original continuation. If the called function is not a procedure, it means that it is replaced with a user-defined closure. In this case, the closure is interpreted by the metalevel interpreter by calling `base-apply` at the metalevel. When the execution of the closure finishes, the result is passed to the original continuation at the original level.

Similarly, the execution of `eval-EM` proceeds by shifting up to the metalevel and executing the argument of `EM` by calling `base-eval` at the metalevel. When the execution finishes, the result is passed back to the original continuation at the original level.

Finally, `base-apply` requires level shifting. Although it is quite similar to `base-apply` in Figure 1 (except for the insertion of `meta-apply`), it has a new additional case where the applied function `op` is not a primitive procedure[3] but still a procedure (the second branch of `cond`). This is the case when the applied function is an evaluator function. In the user-observable interpreter (Figure 1), `base-apply` handles both a primitive and an evaluator function in the same way. In the actual interpreter, however, we need to distinguish them, because application of evaluator functions causes a level to shift down.

The shift down at `base-apply` can be understood as follows. The application of an evaluator function from a user program occurs in two cases: (1) when a user program calls an evaluator function explicitly, like `(base-eval 3 '())`, and (2) when a part of the metalevel interpreter is replaced with a user-defined closure,

[2] To push the environment and continuation, `cons` is used instead of `cons-stream`. This is because `Mcont` must refer to the current value of `Mcont` rather than the value when the tail part of the new `Mcont` is extracted.

[3] The function `primitive-procedure?` returns `#t` (true) if its argument is a primitive procedure, such as `+` and `car`.

```
(define (meta-apply proc-name . args)
  (shift k (shift-up (lambda (mr mk)
    (let ((op (cdr (get proc-name mr))))
      (if (procedure? op)
          (shift-down (k (apply op args))
                      mr mk)
          (let ((x (meta-apply 'base-apply
                               op args mr)))
            (shift-down (k x) mr mk))))))))
(define (eval-EM e r) ; (EM exp)
  (shift k (shift-up (lambda (mr mk)
    (let ((x (meta-apply 'base-eval
                         (car (cdr e)) mr)))
      (shift-down (k x) mr mk))))))
(define (base-apply op args r)
  (cond ((primitive-procedure? op)
         (meta-apply 'unit (apply op args)))
        ((procedure? op) ; evaluator functions
         (shift k
           (shift-down
             (go-up (apply op args))
             (get-global-env r) k)))
        ((and (pair? op)
              (eq? (car op) lambda-tag))
         ... similar to Figure 1 ... )
        (else
         (meta-apply 'my-error
           (list 'Not 'a 'function: op) r))))
(define (go-up x)
  (shift-up (lambda (mr mk) (mk x))))
```

Figure 2. The Black interpreter (level-shifting functions)

and during the execution of that closure, a default compiled evaluator function is called, like the call to `old-eval` from the user-defined `base-eval` shown at the end of Section 3. In the former case, application of an evaluator function means launching a new interpreter below the current level. In the latter case, since the interpretation of the user-defined closure has finished and the execution resumes at the original level, the execution moves from the current level to the level below. In both cases, the level shifts down. To realize this shift down, `base-apply` first captures the current continuation in `k` and pushes it together with the current (global) environment into the metacontinuation (through `shift-down`). It then applies `op` to its argument at the level below. When the execution finishes, the result is returned back to the current level by passing it to `go-up` that returns its argument to the metalevel continuation (also shown in the figure).

4.4 Example

We can now execute the examples shown in the introduction. Here, we will demonstrate another example where we replace the two monadic operators `unit` and `bind`. Remember that the implementation of `my-error` (and hence the behavior of `exit` construct because `eval-exit` depends on `my-error`) was unsatisfactory. There are at least two approaches to remedy the situation. The first one is to introduce an error value and replace the identity monad with the error monad, which we demonstrate in this section. This solution has a benefit that it does not require modification of the Black system itself, but can be implemented within the ordinary user program. On the other hand, the system itself remains unsatisfactory and it leaves us a question whether it is possible at all to construct a reflective system in which abortion is handled more nicely. To address this question, the second solution uses the control operator `shift` to discard the current computation, which we will discuss in the next section.

The first solution is as follows:

```
> (black)
0-0: start
0-1> (EM (begin
       (define error-tag (cons 'error '()))
       (define (raise v) (cons error-tag v))
       (set! bind (lambda (v u)
         (if (and (pair? v)
                  (eq? (car v) error-tag))
           v
           (u v))))
       (set! my-error (lambda (e r) (raise e)))))
0-1: my-error
0-2>
```

We replace the `bind` operator with the one from an error monad and introduce a new monadic operator `raise` to raise an error. The address of the cons cell for `error-tag` is used to distinguish an error value from the ordinary value. If the first argument of `bind` turns out to be an error value, the second argument (continuation of `bind`) is discarded and the error value is returned. After this modification, we have the following interaction:

```
0-2> (* 2 (+ 1 4))
0-2: 10
0-3> ((#t 3) 4)
0-3: ((error) Not a function: #t)
0-4>
```

If the computation does not raise any error, we obtain the result as before. If an error occurs, on the other hand, all the rest of the computation (in the above example, application to 4) is discarded and the error value is returned.

This scenario is much better than the previous one. However, it is still not completely satisfactory.

```
0-4> (exit 0)
0-5: ((error) . 0)
0-6>
```

Even if we want to exit the current level, we can't, because `exit` is simply a variant of `my-error`. Instead of exiting the current level, it prints an error and stays in the same level. To actually exit the current level, we need to treat `my-error` as a level-shifting function.

5. Supporting `exit`

In this section, we consider how to finish the current level in `my-error`. Rather surprisingly, the problem turns out to be not so easy to solve.

5.1 Simple implementation

At first sight, it appears that we could simply and naturally define `my-error` as follows:

```
(define (my-error e r)
  (shift k (shift-up (lambda (mr mk)
    (set-value! 'old-env r mr)
    (set-value! 'old-cont k mr)
    (mk e)))))
```

To exit the current level, we store the current continuation in `k`, shift up one level, and execute the metalevel continuation `mk`. We can even memoise the values of `r` and `k` in the metalevel environment `mr`, so that after exiting to the level above, we can examine the value of `r` through the name `old-env` and resume the aborted

computation by applying old-cont. We then have the following interaction:

```
> (black)
0-0: start
0-1> (exit 0)
1-0: 0
1-1> old-cont
1-1: #<procedure #2>
1-2>
```

The first number in the prompt indicates that we exit to the level 1. We see that the value old-cont is bound to a procedure. It contains the aborted baselevel computation. We can resume it by applying old-cont to a value.

```
1-2> (old-cont 10)
0-1: 10
0-2>
```

Because the execution is now back at the baselevel, the first number of the prompt is 0 again. Furthermore, the second number indicates that the value passed to old-cont becomes the value of (exit 0).

This capability of exiting the current level and visiting the metalevel is particularly useful in the interpreter environment. Rather than issuing a sequence of expressions as an argument to EM, we can simply go up one level, modify the metalevel interpreter as we wish, and go back to the baselevel to see how the modified interpreter works. In fact, in our previous work [1], we treated exit as the main reflective construct.

So far, so good. However, the above definition of my-error leads to a rather subtle anomaly. The above scenario of going up and down works well only until the metalevel interpreter is modified. Suppose that we exit the baselevel again and install tracing into base-eval as we did in Section 3. We can do it without using EM, now:

```
0-2> (exit 0)
1-2: 0
1-3> (define old-eval base-eval)
1-3: old-eval
1-4> (set! base-eval
           (lambda (e r)
             (write e) (newline) (old-eval e r)))
1-4: base-eval
1-5>
```

After going back to the baselevel, a trace is displayed as expected.

```
1-5> (old-cont 0)
0-2: 0
0-3> (* 2 (+ 1 4))
(* 2 (+ 1 4))
*
2
(+ 1 4)
+
1
4
0-3: 10
0-4>
```

However, anomaly arises when we want to exit again.

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

We can no longer exit the level. Why does it happen?

5.2 Level-shifting anomaly

The level-shifting anomaly of not being able to exit the current level once we modify the metalevel interpreter stems from the fact that going up and down are not completely inverse of each other. Although it is easy to show that shift-up and shift-down are inverse of each other, their uses in meta-apply and base-apply are not. There are two cases to consider. First, whenever an evaluator function calls another compiled evaluator function via meta-apply, we make a round trip. Does this going up and down cause any problem? No. After coming back, the state is exactly the same as before, as we can confirm below. Assume that proc-name is bound to a compiled function f.

```
   (meta-apply proc-name . args)
-> (shift k (shift-up (lambda (mr mk)
     (shift-down (k (apply f args)) mr mk))))
-> (shift k (k (apply f args)))
-> (apply f args)
```

At the second step, shifting up followed by shifting down is canceled. At the last step, we used an axiom for shift [10]: (shift k (k M)) is equal to M if k does not occur free in M. The details of the above derivation is not important. What we observe here is that a call to meta-apply reduces correctly to a call to the corresponding compiled function. In other words, the use of meta-apply in this setting is harmless.

The second case is more complicated. It happens when a compiled function calls a user-defined closure and after possible side-effects, the closure calls another compiled function as a tail call. The typical example is the tracing eval:

```
(set! base-eval
      (lambda (e r)
        (write e) (newline) (old-eval e r)))
```

Suppose that this user-defined base-eval is called from a compiled function, *e.g.*, a REP loop. When called, base-eval displays a trace and transfers control to old-eval. In this case, after printing of e is finished, we want the execution to continue at the current level as though old-eval was directly called from the original caller, *i.e.*, the REP loop. In other words, we want to cancel out going up and down needed to print traces.

However, in the current implementation, the original state is not completely recovered. Suppose that proc-name is bound to a user-defined closure op (such as base-eval above).

```
   (meta-apply proc-name . args)
-> (shift k (shift-up (lambda (mr mk)
     (let ((x (meta-apply 'base-apply op args mr)))
       (shift-down (k x) mr mk)))))
```

Assuming that base-apply is not redefined, the execution proceeds by interpreting the body of op. If the body of op calls a compiled function f (such as old-eval above) at the tail position, the execution eventually reaches the second branch of base-apply:

```
-> (shift k (shift-up (lambda (mr mk)
     (let ((x (shift k2
                (shift-down
                  (go-up (apply f args))
                  (get-global-env mr) k2))))
       (shift-down (k x) mr mk)))))
```

At this point, the continuation bound to k2 is (shift-down (k □) mr mk), in other words, "go down and execute k." It is then stored in the metacontinuation by shift-down. As a result, the above expression reduces to:

```
(go-up (apply f args))
```

where the metalevel continuation (stored in the metacontinuation) contains additional frame "go down" at the top of its continuation.

Again, the details of this derivation is not important. The point is that the original call does not reduce to a call to `f` but it is wrapped with "go up" at the end of the current continuation, and the metacontinuation contains "go down" at the beginning. If the execution of `f` finishes normally, there arises no problem. The result is passed to the metalevel and is immediately sent back to the current level. However, if `f` aborts, a problem arises. Even if the current continuation is discarded and the control is transferred to the metalevel, the metalevel continuation contains superfluous frame "go down" at the front. Because of this frame, the execution immediately goes back to the baselevel, prohibiting exit.

5.3 Analysis

Both the "go down" frame in `base-apply` and the "go up" frame in `meta-apply` appear to be necessary. The former is required because the execution of `(apply op args)` in `base-apply` might finish and return a value (rather than calling another compiled function in tail position). To properly pass the result to the current level, the application has to be wrapped by the "go up" frame. The latter is required for a similar reason. If the execution of `(meta-apply 'base-apply op args mr)` in `meta-apply` at the metalevel finishes and returns a value, it has to be passed to the current level continuation `k` after going down.

However, closer inspection of these two functions reveals that going down in `base-apply` is split into two cases. One is when a new level is actually spawned (*e.g.*, a user executes an evaluator function) and the other is when the execution called from the level below is finished and the computation goes back to the level below (*e.g.*, the execution of redefined `base-eval` is finished and `old-eval` is called). The former requires the "go up" frame but the latter does not because it has the "go down" frame at the top, so we could instead cancel out the "go down" frame.

Unfortunately, to distinguish these two cases, we need to peek in the context to see if the closest frame is the "go down" frame. Such an operation is usually not permitted.

5.4 Solution

To avoid level-shifting anomaly, we need to examine the top frame of the system stack. The best way to achieve it would be to support *tail-reflection optimization* considered in [5], similar in spirit to tail-call optimization. However, directly supporting tail-reflection optimization requires modification of the underlying Scheme implementation. Since the optimization is specific to a reflective system, rather than modifying the Scheme implementation, we emulate the tail-reflection optimization in this paper by transforming the whole interpreter into CPS once and for all.

Converting the Black interpreter into CPS spoils some of our original benefits of having reflection in direct style. All the functions that are exposed for user modification have to be written in CPS. However, it appears to be the only solution under the environment where tail-reflection optimization is not supported. The good news is that we have to do it only once when we build a Black interpreter. Once it is done, we are able to reflect on the metalevel interpreter written in direct style.

The CPS transformation is mechanical. We use the following two monadic operators from the continuation monad:

```
(define (munit x) (lambda (k) (k x)))
(define (mbind u v)
  (lambda (k) (u (lambda (x) ((v x) k)))))
```

and expand all the (serious) nested calls using these operators. For example, the result of CPS transformation of `eval-application` is as follows:

```
(define (eval-application e r) ; (f a b c ...)
  (mbind (meta-apply 'eval-list e r)
    (lambda (l0)
      (meta-apply 'bind l0
        (lambda (l)
          (meta-apply 'base-apply
                      (car l) (cdr l) r))))))
```

We are tempted to instantiate `unit` and `bind` to the ones from the continuation monad. We would then obtain a CPS interpreter immediately without modifying the interpreter (except for `unit` and `bind`). It is possible, but it results in a reflective interpreter where the metalevel interpreter is written in direct style but not in monadic style. Because `unit` and `bind` are given a special status, they are no longer visible from user programs. We will not take this approach in this paper to keep the ability to modify monadic operators.

Level-shifting functions are CPS transformed as in Figure 3. Because of the CPS transformation, we do not need `shift` any more, but the current continuation can be captured simply by lambda abstraction. Thus, `(shift k ...)` in Figure 2 is transformed to `(lambda (k) ...)` in Figure 3. Likewise, the context of a serious function call is transformed to an application of the call to its continuation. For example, in `meta-apply`, `(k (apply op args))` is transformed to `((apply op args) k)` and

```
(let ((x (meta-apply 'base-apply op args mr)))
  (shift-down (k x) mr mk))
```

is transformed to:

```
((meta-apply 'base-apply op args mr)
 (lambda (x) (shift-down (k x) mr mk)))
```

The CPS transformation of `k` in `my-error` is a bit complicated. It is basically a composition of `k` and `k2`, but `shift-up` and `shift-down` are inserted to properly adjust levels.

Because contexts are made explicit as continuations, we can now examine the top frame of a continuation by *defunctionalizing* [14] continuations. See Figure 4. We distinguish the frame "go down" `(lambda (x) (shift-down (k x) mr mk))` by representing it as a list of its free variables `(list k mr mk)` (see the last line of `meta-apply` and `eval-EM` and the last part of `my-error` in Figure 4). Whenever a continuation is applied to a value `x`, we use an apply function `(throw k x)` as in `munit` and `my-error`. It checks whether the continuation is a list and if it is, it executes `(shift-down (k x) mk mr)`. Similarly, `mbind` is changed to cope with this new frame.

By representing the "go down" frame as a list, it becomes possible to perform tail-reflection optimization. When we go down a level at the second branch of `base-apply`, we use a special operator `shift-down/go-up`. The role of this operator is to execute its first argument one level below with the "go up" continuation, but if the current continuation is the "go down" frame `(list k2 mr mk)`, it cancels them out and installs `k2` as the continuation of the level below.

With the above optimization, we can execute all the examples shown in this paper (except that the last `(exit 0)` in Section 5.1 is properly handled). We obtained a reflective language where the user-observable metalevel interpreter is written in direct style and which allows us to freely exit the current level.

6. Parser example

In this section, we show as a bigger example how a monadic parser [7] can be implemented by changing the metalevel interpreter.

The idea of a monadic parser is to interpret a context-free gram-

```
(define (meta-apply proc-name . args)
  (lambda (k) (shift-up (lambda (mr mk)
    (let ((op (cdr (get proc-name mr))))
      (if (procedure? op)
          (shift-down ((apply op args) k)
                      mr mk)
          ((meta-apply 'base-apply op args mr)
           (lambda (x)
             (shift-down (k x) mr mk)))))))))
(define (eval-EM e r) ; (EM exp)
  (lambda (k) (shift-up (lambda (mr mk)
    ((meta-apply 'base-eval (car (cdr e)) mr)
     (lambda (x) (shift-down (k x) mr mk)))))))
(define (base-apply op args r)
  (cond ((primitive-procedure? op)
         (meta-apply 'unit (apply op args)))
        ((procedure? op) ; evaluator functions
         (lambda (k)
           (shift-down
             ((apply op args) go-up)
             (get-global-env r) k)))
        ((and (pair? op)
              (eq? (car op) lambda-tag))
         (let ((params        (car (cdr op)))
               (body     (car (cdr (cdr op))))
               (env (car (cdr (cdr (cdr op))))))
           (meta-apply 'eval-eval body
             (extend env params args))))
        (else
         (meta-apply 'my-error
           (list 'Not 'a 'function: op) r))))
(define (go-up x)
  (shift-up (lambda (mr mk) (mk x))))
(define (my-error e r)
  (lambda (k) (shift-up (lambda (mr mk)
    (set-value! 'old-env r mr)
    (set-value! 'old-cont
      (lambda (x) (lambda (k2)
        (shift-up (lambda (mr2 mk2)
          (shift-down (k x) mr2
            (lambda (x)
              (shift-down (k2 x) mr2 mk2)))))))
      mr)
    (mk e)))))
```

Figure 3. The Black interpreter in CPS (level-shifting functions)

mar as a program. For example, consider the following grammar representing addition and multiplication over numbers.

$$\begin{array}{lcllcl} E & ::= & T\ E' & T & ::= & N\ T' \\ E' & ::= & +\ T\ E' \mid \epsilon & T' & ::= & *\ N\ T' \mid \epsilon \\ & & & N & ::= & \textit{number} \end{array}$$

Following this grammar, we write a parser program as in Figure 5. The functions `e`, `e2`, `t`, `t2`, and `num` correspond to the non-terminals E, E', T, T', and N, respectively. When called, these functions parse an input string (implicitly passed around as a state), and return all the possible parse trees together with the unparsed strings. Whether an input string is completely parsed by the grammar can be judged by checking the unparsed string: if it is an empty string, the input is completely parsed (see `finish`).

When defining these functions, we can use several monadic operators provided by the parser monad. The choice in the grammar is handled by `amb`. It tries to parse both the alternatives. When

```
(define (throw k x)
  (if (pair? k)
    (let ((k2 (car k))
          (mr (car (cdr k)))
          (mk (car (cdr (cdr k)))))
      (shift-down (throw k2 x) mr mk))
    (k x)))
(define (munit x) (lambda (k) (throw k x)))
(define (mbind u v) (lambda (k)
  (if (pair? k)
    (let ((k2 (car k))
          (mr (car (cdr k)))
          (mk (car (cdr (cdr k)))))
      (u (list
           (lambda (x) (shift-up (lambda (mmr mmk)
             ((v x) (list k2 mmr mmk)))))
           mr mk)))
    (u (lambda (x) ((v x) k))))))
(define (meta-apply proc-name . args)
  (lambda (k) (shift-up (lambda (mr mk)
    (let ((op (cdr (get proc-name mr))))
      (if (procedure? op)
          (shift-down ((apply op args) k)
                      mr mk)
          ((meta-apply 'base-apply op args mr)
           (list k mr mk)))))))) ; pass a list
(define (eval-EM e r) ; (EM exp)
  (lambda (k) (shift-up (lambda (mr mk)
    ((meta-apply 'base-eval (car (cdr e)) mr)
     (list k mr mk)))))) ; pass a list
(define (base-apply op args r)
  (cond ((primitive-procedure? op)
         (meta-apply 'unit (apply op args)))
        ((procedure? op) ; evaluator functions
         (lambda (k)
           (shift-down/go-up ; optimize
             (apply op args)
             (get-global-env r) k)))
        ((and (pair? op)
              (eq? (car op) lambda-tag))
         ... the same as Figure 3 ... )
        (else
         (meta-apply 'my-error
           (list 'Not 'a 'function: op) r))))
(define (my-error e r)
  (lambda (k) (shift-up (lambda (mr mk)
    (set-value! 'old-env r mr)
    (set-value! 'old-cont
      (lambda (x) (lambda (k2)
        (shift-up (lambda (mr2 mk2)
          (shift-down (throw k x)
                      mr2 (list k2 mr2 mk2))))))
      mr)                ; pass a list
    (mk e)))))
(define (shift-down/go-up code r k)
  (if (pair? k)
    (let ((k2 (car k))
          (mr (car (cdr k)))
          (mk (car (cdr (cdr k)))))
      (shift-down (code k2) mr mk))
    (shift-down (code go-up) r k)))
```

Figure 4. The defunctionalized Black interpreter with tail-reflection optimization

```
(define (sat p)
  (let ((lst (read-state)))
    (cond ((null? lst) (none))
          ((p (car lst)) (write-state! (cdr lst))
                         (car lst))
          (else (none)))))
(define (item c) (sat (lambda (x) (eq? x c))))
(define (num) (sat number?))
(define (e)  ; E  = T E2
  (let* ((x (t)) (lst (e2)))
    (if (null? lst) x (cons '+ (cons x lst)))))
(define (e2) ; E2 = + T E2 | ϵ
  (amb (let* ((i (item '+)) (x (t)) (lst (e2)))
         (cons x lst))
       '()))
(define (t)  ; T  = Num T2
  (let* ((x (num)) (lst (t2)))
    (if (null? lst) x (cons '* (cons x lst)))))
(define (t2) ; T2 = * Num T2 | ϵ
  (amb (let* ((i (item '*)) (x (num)) (lst (t2)))
         (cons x lst))
       '()))

(define (finish result)
  (if (null? (read-state)) result (none)))
(define (parse grammar lst)
  (write-state! lst)
  (finish (grammar)))
```

Figure 5. Monadic parser (baselevel program)

parsing fails, `none` is used. To obtain the input string, we use `read-state`, and the input string is initialized and updated (*e.g.*, parsed string is removed) by `write-state!`.

Compared to the standard monadic parsers, the parser presented here does not need monadic programming. Since the monadic operators are provided by changing the metalevel interpreter (to be explained soon), the ordinary constructs such as `let*` and `if` are interpreted in a monadic way. Thus, the surface program does not have to mention monads at all.

The monadic operators can be implemented easily. After exiting the current level, we load the program in Figure 6. We first replace `unit` and `bind` with the ones from the parser monad:

$$M\ A = \mathit{String} \rightarrow (A \times \mathit{String})\ \mathit{List}$$

We represent the input string as a list of symbols. The `start` operator initiates the monadic computation by passing the initial string. It is used in the REP loop, whose user-observable definition is as follows:

```
(define (init-cont env level turn answer)
  (write level) (write '-) (write turn)
  (display ": ") (write answer) (newline)
  (write level) (write '-) (write (+ turn 1))
  (display "> ")
  (let ((ans (start (base-eval (read) env))))
    (init-cont env level (+ turn 1) ans)))
```

The evaluator functions for the new monadic operators follows in Figure 6. They are registered as special forms by inserting case branches before `eval-application`, in other words, before they are treated as ordinary function calls. We then launch the parser interpreter by calling `init-cont`, with "`parser`" as the name of the level. Suppose that the functions in Figure 5 and Figure 6 are saved in files "`parser-meta.scm`" and "`parser.scm`", respectively.

```
(set! unit (lambda (x)
             (lambda (lst) (list (cons x lst)))))
(set! bind (lambda (u v) (lambda (lst)
  (apply append
    (map (lambda (p) ((v (car p)) (cdr p)))
         (u lst))))))
(set! start (lambda (x) (x '())))

(define (eval-read-state e r) ; (read-state)
  (lambda (lst) ((unit lst) lst)))
(define (eval-write-state! e r);(write-state! exp)
  (bind (base-eval (car (cdr e)) r)
        (lambda (v)
          (lambda (lst) ((unit #f) v)))))
(define (eval-amb e r) ; (amb e1 e2)
  (lambda (lst) (append
    ((base-eval (car (cdr e)) r) lst)
    ((base-eval (car (cdr (cdr e))) r) lst))))
(define (eval-none e r) ; (none)
  (lambda (lst) '()))
(define old-eval-application eval-application)
(set! eval-application (lambda (e r)
  (cond ((eq? (car e) 'read-state)
         (eval-read-state e r))
        ((eq? (car e) 'write-state!)
         (eval-write-state! e r))
        ((eq? (car e) 'amb)  (eval-amb e r))
        ((eq? (car e) 'none) (eval-none e r))
        (else (old-eval-application e r)))))
```

Figure 6. Monadic parser (metalevel modification/extension)

```
> (black)
0-0: start
0-1> (exit 0)
1-0: 0
1-1> (load "parser-meta.scm")
1-1: done
1-2> (init-cont init-env 'parser 0 'start)
parser-0: start
parser-1> (load "parser.scm")
parser-1: ((done))
parser-2> (parse e '(2 * 3 + 4 * 5 * 6 + 7))
parser-2: (((+ (* 2 3) (* 4 5 6) 7)))
parser-3> (exit 0)
1-2: 0
1-3>
```

The above grammar is unambiguous, but if we implement ambiguous grammar, we will obtain all the possible parse trees.

Although we launched a REP loop in the above example, more practical approach would be to call `(start (base-eval exp init-env))` where `exp` is the baselevel parser program (Figure 5) followed by the main expression to initiate parsing (such as `(parse e '(2 * 3 + 4 * 5 * 6 + 7))` above). We could then use the monadic parser whenever we need it, but stay in the ordinary interpreter otherwise. Furthermore, we could consider optimizing `(start (base-eval exp init-env))` by specializing [9] (modified) `base-eval` with respect to `exp`, *i.e.*, compiling `exp` under the parser semantics defined by the modified `base-eval`. Because we could resolve the level-shifting anomaly and fix the basic framework of the reflective language, it becomes possible to think of such interesting optimization.

7. Discussion and future direction

7.1 `EM` vs. `exit`

In this paper, we have introduced two reflective constructs, `EM` and `exit`. If we used only `EM`, we would never encounter the level-shifting anomaly, because we cannot observe the superfluous "go up" and "go down" frames. Then, do we need `exit` at all? We believe yes at least for two reasons. The process of writing a reflective program consists of two parts: a metalevel program followed by a baselevel program. If `exit` was not provided, we have to wrap metalevel programs with `EM` all the time, which is quite cumbersome. Furthermore, we make mistakes. Proper handling of errors, which is realized by correct handling of the exit mechanism, is essential in programming in an interpreter environment.

7.2 Direct style vs. CPS

In the previous work [2], we have already shown how to build a reflective system where the metalevel interpreter is written in CPS. Because we can transform any program into CPS, all we can do with a direct-style Black can be done in the CPS Black. Then, do we really need a reflective system whose metalevel interpreter is written in direct style? We believe yes, because there is an important difference between the two systems. In CPS Black, we *must* write the metalevel interpreter in CPS. Otherwise, the `exit` mechanism breaks down. In CPS Black, `exit` is implemented as throwing away the current continuation. This is possible only when the metalevel interpreter is properly written in CPS. It does not work any more if we redefine evaluator functions in direct style. In other words, the behavior of `exit` in CPS Black depends on the user's writing CPS. In direct-style Black, such danger does not exist. To put differently, the DS Black makes it possible for the first time to perform `exit` at all times.

7.3 Interpreted code vs. compiled code

One of the motivations of this work is to establish a foundation for efficient implementation of reflective languages. This goal is not yet achieved. Rather, we have set up the basis on which efficient implementation is tried. In our previous work on CPS Black, we have already mentioned the compilation framework of reflective languages: converting interpreted code into compiled code via partial evaluation [9] of the metalevel interpreter with respect to baselevel program. Since the metalevel interpreter (the language semantics) is modifiable, we cannot construct a fixed compiler because it depends on the particular language semantics. However, we have not been able to tackle compilation, because so far we suffered from the level-shifting anomaly and the restriction that the metalevel interpreter has to be written in CPS. Now that we have solved these problems, we are ready to consider partial evaluation seriously. Another promising approach would be to try traced-based compilation framework successfully applied to the PyPy project [3] to reduce interpretive overhead.

8. Conclusion

In this paper, we described the reflective language Black whose metalevel interpreters are written in direct style. Implementation of the reflective construct `EM` was possible by the manipulation of metacontinuations, but implementation of seemingly simple construct `exit` turned out to require tail-reflection optimization. We have achieved the same effect by CPS transforming the metalevel interpreter once and for all. Although the Black interpreter is now written in CPS, the user-observable interpreter is in direct style. Thus, user programs can reflect on the direct-style metalevel interpreter. We hope that the resulting system can be a base platform for the efficient execution of reflective programs in the future.

Acknowledgments

Thanks to the anonymous reviewers for interesting comments. This work was partly supported by JSPS Grant-in-Aid for Scientific Research (C) 22500025.

References

[1] Asai, K. "Reflecting on the Metalevel Interpreter Written in Direct Style," Presented at the International Lisp Conference 2003 (ILC 2003), New York City, 12 pages (October 2003).

[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] Bolz, C. F., A. Cuni, M. Fijalkowski, and A. Rigo "Tracing the Meta-Level: PyPy's Tracing JIT Compiler," *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pp. 18–25 (July 2009).

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

[5] 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).

[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] Hutton, G., and E. Meijer "Monadic Parsing in Haskell," *Journal of Functional Programming*, Vol. 8, No. 4, pp. 437–444, Cambridge University Press (July 1998).

[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] Jones, N. D., C. K. Gomard, and P. Sestoft *Partial Evaluation and Automatic Program Generation*, New York: Prentice-Hall (1993).

[10] Kameyama, Y., and M. Hasegawa "A Sound and Complete Axiomatization of Delimited Continuations," *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pp. 177–188 (August 2003).

[11] Kiczales, G., J. des Rivières, and D. G. Bobrow *The Art of the Metaobject Protocol*, Cambridge: MIT Press (1991).

[12] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J-M. Loingtier, and J. Irwin "Aspect-Oriented Programming," *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, pp. 220–242 (1997).

[13] Masuhara, H., and A. Yonezawa "Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Language," *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'98), LNCS 1445*, pp. 418–439 (July 1998).

[14] Reynolds, J. C. "Definitional Interpreters for Higher-Order Programming Languages," *Proceedings of the ACM National Conference*, Vol. 2, pp. 717–740, (August 1972), reprinted in *Higher-Order and Symbolic Computation*, Vol. 11, No. 4, pp. 363–397, Kluwer Academic Publishers (December 1998).

[15] 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).

[16] Verwaest, T., C. Bruni, D. Gurtner, A. Lienhard, and O. Niestrasz "PINOCCHIO: Bringing Reflection to Life with First-Class Interpreters," *Proceedings of the ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '10)*, pp. 774–789, (October 2010).

[17] 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).