, , 1–40 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Duplication and Partial Evaluation

— For a Better Understanding of Reflective Languages —

KENICHI ASAI asai@is.s.u-tokyo.ac.jp
Department of Information Science, Faculty of Science, The University of Tokyo, 7-3-1 Hongo, Bunkyo-Ku, Tokyo, 113, Japan

SATOSHI MATSUOKA matsu@ipl.t.u-tokyo.ac.jp
Department of Mathematical Engineering, Faculty of Engineering, The University of Tokyo, 7-3-1 Hongo, Bunkyo-Ku, Tokyo, 113, Japan

AKINORI YONEZAWA yonezawa@is.s.u-tokyo.ac.jp
Department of Information Science, Faculty of Science, The University of Tokyo, 7-3-1 Hongo, Bunkyo-Ku, Tokyo, 113, Japan

Received October 10, 1993; Revised August 6, 1994, March 30, 1995

Editor: Daniel P. Friedman and Akinori Yonezawa

Abstract. This paper presents a general implementation framework for reflective languages. It allows us to systematically build reflective languages which have the following favorable properties: (1) user programs are allowed to access and change (parts of) metalevel interpreters, (2) reflective facilities are available at every level, (hence there exists conceptually an infinite tower of interpreters), and (3) the interpreter runs as efficiently as the conventional (directly implemented) metacircular interpreter when reflection is not used. Our scheme is divided into three stages. First, we define interpreters which give the operational semantics of each level, and conceptually construct the infinite tower of these interpreters. They are then *duplicated* to obtain directly executed interpreters, while introducing double interpretation to maintain redefinability of interpreters. Finally, partial evaluation is employed to collapse the double interpretation into single interpretation. We illustrate our scheme by implementing a particular reflective language called *Black* in Scheme, but it is general enough to be applied to other reflective languages. The paper gives the complete Scheme implementation of Black and demonstrates some examples. We also show how a system with the delta abstraction introduced by Blond can be constructed in our framework.

Keywords: reflection, partial evaluation, metacircular interpreter, direct execution, interpretation, infinite regression

"Black is Black" — Los Bravos

1. Introduction

Reflective programming languages provide high flexibility and extensibility by allowing user programs to access and change their language implementation (and thereby semantics) from within themselves. To change the language semantics in conventional non-reflective languages, we need to fully understand the low level

implementation details and carefully alter their implementation (either an interpreter or a compiler), which is usually written in lower level languages. Reflective languages allow such modifications within the same language framework using high level abstractions without understanding low level implementation. Because of its flexibility and extensibility, reflection is also important for practical purposes, i.e., as a basis for "open implementation"[9].

An important characteristic of reflective languages is their *tower* structure. In reflective languages, user programs at the *baselevel* (or level 0) are interpreted by the interpreter running at the level above (level 1, or *metalevel*), which is in turn interpreted by the interpreter running at level 2, and so on. Conceptually, there is an infinite tower of interpreters. Reflective languages permit user programs to execute programs at any level in the tower, possibly accessing (introspecting) or modifying interpreters of upper levels.

However, the infinite tower structure makes it difficult to implement reflective languages. In conventional Lisp-based reflective languages, such as Brown[5][13] and Blond[2], the tower is simulated by *lazily* creating interpreters which are *directly executed* by machine, instead of by interpretation at the level above. Although the use of directly executed interpreters is preferable for efficiency, it becomes impossible for user programs to alter (parts of) the interpreters selectively, since they are already in machine code form. For example, suppose we want to modify the metalevel interpreter so that it prints out all the expressions it evaluates. If the structure and implementation of the metalevel interpreter were available to user programs, we could achieve this just by inserting a print function at an appropriate place in the interpreter code. Such modifications are not possible in Brown and Blond since the interpreter is directly executed and available only as a single monolithic function called *meaning*.

Jefferson and Friedman, on the other hand, presented a Simple Reflective Interpreter $\mathcal{I}_R$[7], in which user programs are allowed to access and change the metalevel interpreters. Because in $\mathcal{I}_R$, the interpreter running at each level is implemented as ordinary user-defined functions, and *actually interpreted* by the interpreter one level above, we can freely redefine it to change its subsequent behavior. However, since $\mathcal{I}_R$ is implemented by executing a metacircular interpreter on top of itself, it has two drawbacks: (1) it is extremely inefficient due to double (triple, etc.) interpretation, and (2) it requires the construction of an infinite tower if we want to achieve reflection at every level.

To be highly flexible and extensible, reflective languages should fulfill the following requirements:

1. User programs are allowed to access and change (parts of) metalevel interpreters.

2. Reflective facilities are available at every level.

3. The interpreter runs efficiently when reflection is not used, and reasonably well even if parts of it are modified.

In this paper, we present a general framework for the implementation of reflective languages that satisfy these three requirements. To be more specific, the scheme is divided into three stages. First, we define interpreters that give the operational semantics of each level, and conceptually construct the infinite tower of these interpreters. In conventional reflective languages, metalevel interpreters observed by user programs are often confused with the actual implementation. The clear separation between the "user observable interpreters" and "actually implemented interpreters" plays an important role in a better understanding of reflective languages. We then *duplicate* the interpreters and obtain directly executed interpreters to avoid infinite regression. To maintain redefinability of interpreters, however, the infinite tower is not constructed from them alone. Rather, each level is (initially) made to consist of a directly executed interpreter together with an extra metacircular interpreter, tentatively introducing inefficient double interpretation. Finally, partial evaluation[8] is employed to collapse the double interpretation into single interpretation. The resulting tower of interpreters is implementable in finite resources by lazy creation, executes user programs almost as efficiently as a single directly executed interpreter, but still allows modification of the component interpreters. In practice, metacontinuations are used to create interpreters lazily.

Although the obtained tower is still an approximation of the original reflective tower, it exhibits reasonable and favorable characteristics:

- It satisfies the above three requirements, achieving sufficient flexibility and efficiency.
- The approximated fragments of the interpreter can be naturally understood as directly executed compiled code rather than interpreted code.

The faithful implementation of the infinite tower imposes inefficiency due to double (triple, etc.) interpretation unless dynamic compilation is used. Although dynamic compilation using a partial evaluator is an interesting topic, it is still an open research area and we do not discuss it further in this paper. Rather, we aim at constructing a more static system, in which both flexibility and efficiency are obtained without dynamic compilation. We illustrate our scheme using a particular reflective language called *Black*, but the scheme itself is general enough to be applied to other reflective languages. In fact, the *delta* abstraction introduced by Blond[2] is shown to be easily implementable in our framework.

This paper is organized as follows. The next section introduces the Black language and its metalevel interpreter that defines the operational semantics of each level. Sections 3 and 4 discuss duplication and partial evaluation, respectively. The actual Scheme implementation of Black is presented in Section 5, followed by some example Black program executions in Section 6. In Section 7, we describe how a system with the delta abstraction can be implemented in our framework. Related work is discussed in Section 8, and the paper concludes in Section 9. The Appendix shows the Black implementation code not covered in Section 5. Together, they comprise the complete Scheme interpreter of Black, which is also obtainable via `ftp://camille.is.s.u-tokyo.ac.jp`(133.11.12.1)`/pub/black/black.tar.Z`.

2. The Reflective Language Black

Black is a Scheme-based reflective language with an infinite tower of interpreters as in Brown and Blond. In contrast to Brown and Blond, however, it does not have a *reifier* that takes a baselevel expression, environment, and continuation. Instead, it has a special reflective construct `(exec-at-metalevel E)`, that evaluates the expression `E` one level above. Since `E` is executed at the same level as the metalevel interpreter, `E` is allowed to access and change the metalevel interpreter as in $\mathcal{I}_R$. By nesting `exec-at-metalevel`, we can access and change the interpreter of any level.

2.1. The Metalevel Interpreters Observed by User Programs

To describe the operational view of the language Black, we first define the interpreters that give the initial operational semantics of each level. Although they are standard continuation passing style (CPS) Scheme metacircular interpreters (except for a few differences mentioned below), we go through and examine the definition in detail. They not only define the operational semantics of each level, but also show user programs how metalevel interpreters are constructed, giving a way to access and change them. Note that the interpreter shown in this section is not the actual implementation of Black, but the *user observable one*. The actual implementation (presented in Section 5) executes user programs in such a way that they can be regarded as being executed by the tower of interpreters presented here.

The interpreter of each level consists of functions (and variables) shown in Figures 1, 2, 3, and Appendices A.1 and A.2. We call them *evaluator functions*, as they collectively comprise a whole evaluator. The main function, named `base-eval` to distinguish it from `eval` in Scheme, and other standard functions are defined in Figure 1. The notable characteristics are that `base-eval` handles the `exec-at-metalevel` special form, and that lambda-closures are represented as lists using a unique `cons` cell (named `lambda-tag`) as a tag.

Figure 2 shows functions that potentially cause a levelshift, and their related variables. The function `eval-EM`[1] is called from `base-eval` when the expression is an `exec-at-metalevel` special form, and evaluates its body at metalevel *using (a primitive variant*[2] *of)* `exec-at-metalevel` *itself.* In other words, reflection is employed to explain reflection. This is a natural consequence, because we are now defining the semantics of the language of each level, not the semantics of the Black language itself. The execution of `exec-at-metalevel` causes a levelshift; since levelshifting operations cannot be understood within a single-level semantics, they are explained in a 'magical' way using `primitive-EM` in the interpreter of each level.

`Base-apply` applies a function to its arguments. It might initially seem that it does not cause a levelshift, but in fact it sometimes does. Intuitively speaking, a downward levelshift occurs, when directly executed (compiled) evaluator functions

```
(define (base-eval exp env cont)
  (cond ((number? exp)                      (cont exp))
        ((boolean? exp)                     (cont exp))
        ((string? exp)                      (cont exp))
        ((symbol? exp)                      (eval-var exp env cont))
        ((eq? (car exp) 'quote)             (eval-quote exp env cont))
        ((eq? (car exp) 'if)                (eval-if exp env cont))
        ((eq? (car exp) 'set!)              (eval-set! exp env cont))
        ((eq? (car exp) 'lambda)            (eval-lambda exp env cont))
        ((eq? (car exp) 'begin)             (eval-begin exp env cont))
        ((eq? (car exp) 'exec-at-metalevel) (eval-EM exp env cont))
        ((eq? (car exp) 'exit)              (eval-exit exp env cont))
        (else (eval-application exp env cont))))
(define (eval-var exp env cont)
  (let ((pair (get exp env)))
    (if (pair? pair)
        (cont (cdr pair))
        (my-error (list 'eval-var: 'unbound 'variable: exp) env cont))))
(define (eval-quote exp env cont) (cont (car (cdr exp))))
(define (eval-if exp env cont)
  (base-eval (car (cdr exp)) env
             (lambda (pred)
               (if pred (base-eval (car (cdr (cdr exp))) env cont)
                        (base-eval (car (cdr (cdr (cdr exp)))) env cont)))))
(define (eval-set! exp env cont)
  (let ((var (car (cdr exp)))
        (body (car (cdr (cdr exp)))))
    (base-eval body env
               (lambda (data)
                 (let ((pair (get var env)))
                   (if (pair? pair)
                       (begin (set-value! var data env)
                              (cont var))
                       (my-error (list 'eval-set!: 'unbound 'variable var)
                                 env cont)))))))
(define lambda-tag (cons 'lambda 'tag))
(define (eval-lambda exp env cont)
  (let ((lambda-body (cdr (cdr exp)))
        (lambda-params (car (cdr exp))))
    (cont (list lambda-tag lambda-params lambda-body env))))
(define (eval-begin exp env cont)
  (eval-begin-body (cdr exp) env cont))
(define (eval-begin-body body env cont)
  (define (eval-begin-local body)
    (if (null? (cdr body))
        (base-eval (car body) env cont)
        (base-eval (car body) env (lambda (x) (eval-begin-local (cdr body))))))
  (if (null? body)
      (my-error '(eval-begin-body: null body) env cont)
      (eval-begin-local body)))
```

Figure 1. Standard functions observed by user programs

```
(define (eval-exit exp env cont)
  (base-eval (car (cdr exp)) env (lambda (x) (my-error x env cont))))
(define (eval-list exp env cont)
  (if (null? exp)
      (cont '())
      (base-eval (car exp) env
                 (lambda (val1)
                   (eval-list (cdr exp) env
                              (lambda (val2) (cont (cons val1 val2))))))))
(define (eval-application exp env cont)
  (eval-list exp env (lambda (l) (base-apply (car l) (cdr l) env cont))))
(define (eval-map fun lst env cont)
  (if (null? lst)
      (cont '())
      (base-apply fun (list (car lst)) env
                  (lambda (x) (eval-map fun (cdr lst) env
                                        (lambda (y) (cont (cons x y))))))))
```

Figure 1. Standard functions observed by user programs (continued)

```
(define (eval-EM exp env cont)
  (cont (primitive-EM (car (cdr exp)))))
(define (base-apply operator operand env cont)
  (cond ((procedure? operator)
         (cond ((eq? operator map)
                (eval-map (car operand) (car (cdr operand)) env cont))
               ((eq? operator scheme-apply)
                (base-apply (car operand) (car (cdr operand)) env cont))
               ((pair? (memq operator primitive-procedures))
                (cont (scheme-apply operator operand)))
               (else ; evaluator functions
                (cont (scheme-apply operator operand)))))
        ((and (pair? operator)
              (eq? (car operator) lambda-tag))
         (let ((lambda-params        (car (cdr operator)))
               (lambda-body     (car (cdr (cdr operator))))
               (lambda-env (car (cdr (cdr (cdr operator))))))
           (eval-begin-body lambda-body
                            (extend lambda-env lambda-params operand)
                            cont)))
        (else
         (my-error (list 'Not 'a 'function: operator) env cont))))
(define old-env 0)
(define old-cont 0)
(define (my-error exp env cont)
  (set! old-env env)
  (set! old-cont cont)
  exp)
```

Figure 2. Levelshifting functions observed by user programs

```
(define (init-cont env level turn cont)
  (cont (lambda (answer)
          (write level) (write '-) (write turn) (display ": ")
          (primitive-write answer) (newline)
          (write level) (write '-) (write (+ turn 1)) (display "> ")
          (base-eval (read) env
                     (lambda (ans) (init-cont env level (+ turn 1)
                                              (lambda (cont) (cont ans))))))))
(define (run env level answer)
  (init-cont env level 0 (lambda (cont) (cont answer))))
(define init-env (list (list
  (cons 'car        car)        (cons 'cdr              cdr)
  (cons 'cons       cons)       (cons 'list             list)
  (cons 'pair?      pair?)      (cons 'null?            null?)
  (cons 'not        not)        (cons 'eq?              eq?)
  (cons 'eqv?       eqv?)       (cons 'equal?           equal?)
  (cons 'set-car!   set-car!)   (cons 'set-cdr!         set-cdr!)
  (cons 'append     append)     (cons 'newline          newline)
  (cons 'read       read)       (cons 'write            primitive-write)
  (cons '+ +)       (cons '- -) (cons '* *)             (cons '/ /)
  (cons '= =)       (cons '> >) (cons '< <)
  (cons 'quotient   quotient)   (cons 'remainder        remainder)
  (cons 'number?    number?)    (cons 'symbol?          symbol?)
  (cons 'boolean?   boolean?)   (cons 'string?          string?)
  (cons 'memq       memq)       (cons 'length           length)
  (cons 'assq       assq)       (cons 'procedure?       primitive-procedure?)
  (cons 'map        map)        (cons 'scheme-apply     scheme-apply)

  (cons 'empty-env  empty-env)  (cons 'make-pairs       make-pairs)
  (cons 'extend     extend)     (cons 'set-value!       set-value!)
  (cons 'get        get)        (cons 'define-value     define-value)
  (cons 'copy       copy)       (cons 'get-global-env   get-global-env)

  (cons 'base-eval  base-eval)  (cons 'eval-var         eval-var)
  (cons 'eval-quote eval-quote) (cons 'eval-if          eval-if)
  (cons 'lambda-tag lambda-tag) (cons 'eval-lambda      eval-lambda)
  (cons 'eval-begin eval-begin) (cons 'eval-begin-body  eval-begin-body)
  (cons 'eval-set!  eval-set!)  (cons 'eval-EM          eval-EM)
  (cons 'eval-exit  eval-exit)  (cons 'eval-application eval-application)
  (cons 'eval-list  eval-list)  (cons 'base-apply       base-apply)
  (cons 'my-error   my-error)   (cons 'eval-map         eval-map)

  (cons 'init-env   0)          ; to be filled later
  (cons 'init-cont  init-cont)  (cons 'run            run)
  (cons 'old-env    old-env)    (cons 'old-cont       old-cont)
  (cons 'primitive-procedures   primitive-procedures)
)))
(define-value 'init-env (copy init-env) init-env)
```

Figure 3. Bootstrapping functions observed by user programs

are applied, and a new level is spawned. This is why the first clause of the top-level `cond` in `base-apply` is not simply written as[3]:

```
((procedure? operator) (cont (scheme-apply operator operand)))
```

Although we could write as above in the user observable interpreter, we will later need to treat higher-order functions and evaluator functions separately (the latter is distinguished by the last `else` clause, which does not exist in a standard metacircular interpreter) in the actual implementation. `Primitive-procedures`, and other primitive functions that have to be redefined are shown in Appendix A.1.

`My-error` is used to report an error. It is also called from `eval-exit` in Figure 1. It simply ignores its current continuation and returns `exp`, thus terminating the level. Before the termination, it stores the current environment and continuation into `old-env` and `old-cont`, respectively, so that users can later resume the execution by typing (`old-cont` *value*).

Finally, Figure 3 shows bootstrapping functions and an initial environment. Level n interpreter can be created by executing (`run init-env` n `'start`), which starts a read-eval-print loop defined in `init-cont`. An environment is represented as a list of frames (association lists). The initial environment `init-env` contains only one frame, which holds all the primitive functions, as well as evaluator functions that define the level below. In `init-env`, the name `init-env` is temporarily bound to 0, and later at the last line of the figure, it is bound to (the copy of) `init-env` itself. The other thing to notice in `init-env` is the two primitives `write` and `procedure?`: to treat lambda closures appropriately, they are bound to `primitive-write` and `primitive-procedure?`(presented in Appendix A.1), respectively. Environment manipulation functions are shown in Appendix A.2. Destructive operations on lists (`set-car!` and `set-cdr!`) are used to realize baselevel side-effects (see `set-value!` and `define-value`).

2.2. The Structure of the Reflective Tower

Given the definition of the interpreter of each level, the overall tower structure of Black is depicted as in Figure 4(a). The thick lines in the figure connect the functions that call each other, and a thin arrow indicates that the source interprets the target. Initially, a user program `E` runs at level 0 and is interpreted by the level 1 interpreter, which in turn is interpreted by the level 2 interpreter, and so on. Now, suppose that we execute a program containing (`exec-at-metalevel E'`) at level 0. The body `E'` is then executed at level 1 by the level 2 interpreter as illustrated in Figure 4(b). Since both `E'` and evaluator functions of level 1 are uniformly interpreted as ordinary user-defined functions by the level 2 interpreter, the level 1 interpreter is visible from `E'` via the level 2 environment. Thus, we can freely access or change them with their symbolic names `base-eval`, `eval-var`, etc. Let us denote the level n interpreter by $\mathtt{L}_n^{env_n}(\cdot)$ (omitting continuations for simplicity). Then, the Black tower executing `E` can be depicted by an infinite tower of $\mathtt{L}_n^{env_n}$ as in Figure 4(c).

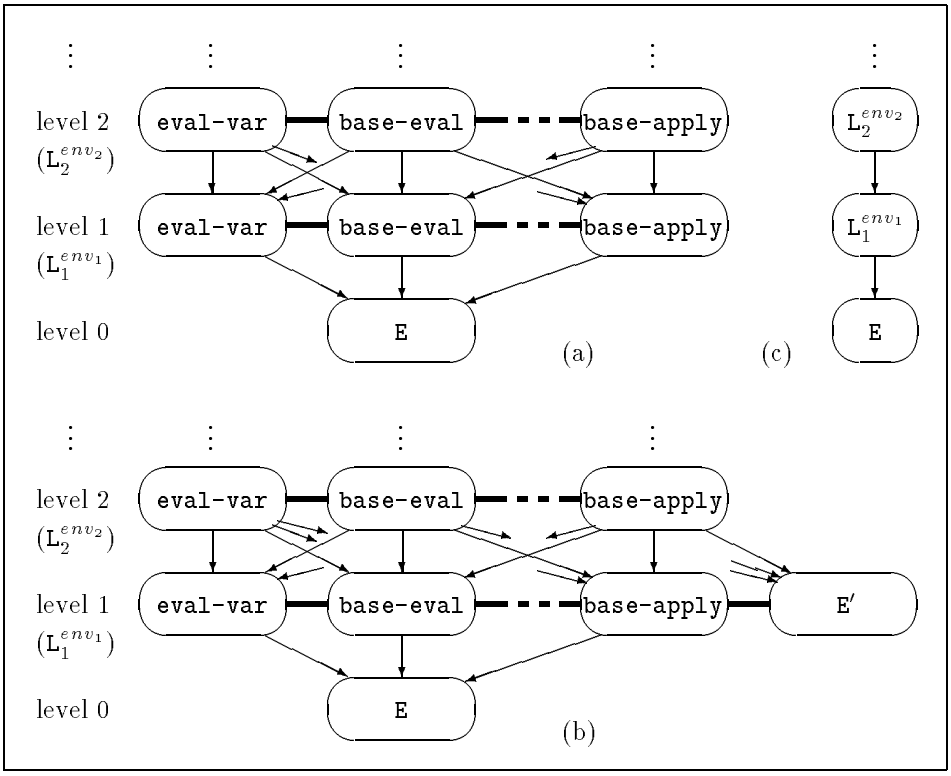

Figure 4. The tower structure of Black

2.3. Notes on Metacircularity

Strictly speaking, the interpreter presented here is not metacircular for two reasons. One is the use of `primitive-EM`, which is not a part of the defined language. The situation could be avoided, if we took `primitive-EM` itself as the reflective construct of Black, rather than `exec-at-metalevel`. This could be realized by defining the following function:

```
(define (eval-primitive-EM exp env cont)
  (base-eval (car (cdr exp)) env
             (lambda (val) (cont (primitive-EM val)))))
```

and replacing the corresponding clause in `base-eval`. We did not do this, however, since `primitive-EM` inherently introduces a large amount of dynamic behavior into the language. `Primitive-EM` allows us to dynamically construct metalevel code and install it dynamically, while `exec-at-metalevel` allows only dynamic installation of statically known code. Although the former causes no problems in constructing

an interpreter, the compiler for the language will require dynamic compilation techniques. Thus, we employed `exec-at-metalevel`, retaining the possibility of static compilation.

The other reason for non-metacircularity is in the lack of definitions for some special forms, such as `cond`, `define`, `let`, `and`, and `or`. They can be easily programmed, and are omitted here for brevity.

2.4. An Example Program

Now that we know the structure of the interpreter at each level, we can easily modify or extend the language itself. This is achieved by allowing user programs access to a set of functions comprising the interpreter. Let us consider an example program that changes the metalevel interpreter using `exec-at-metalevel`. Consider the following program:

```
(exec-at-metalevel (let ((old-eval base-eval))
                     (set! base-eval (lambda (exp env cont)
                                       (write exp) (newline)
                                       (old-eval exp env cont)))))
```

When this program is executed at level 0, the expression `(let ...)` is executed at level 1, where `base-eval`, `eval-var`, and other evaluator functions are already defined. The function `base-eval` is therefore replaced with the closure above that displays the expression before evaluating it. If other functions (`eval-if`, for example) call `base-eval` after this change, it is this new `base-eval` that is called. Thus, after executing the above program, we obtain the following behavior:

```
0-1> (cons 1 2)
(cons 1 2)
cons
1
2
0-1: (1 . 2)
```

Because the level n interpreter can be regarded as defining the operational semantics of the level $n-1$ language, Black effectively allows us to dynamically modify the operational semantics of the language.

3. Duplication

The tower of interpreters presented in the previous section is unimplementable as shown, since every interpreter needs to be interpreted by the interpreter one level above, causing infinite regression. To implement it with finite resources, we have to terminate the regression somewhere, and run an interpreter as executable code instead of interpreting it using the interpreter one level above. Although

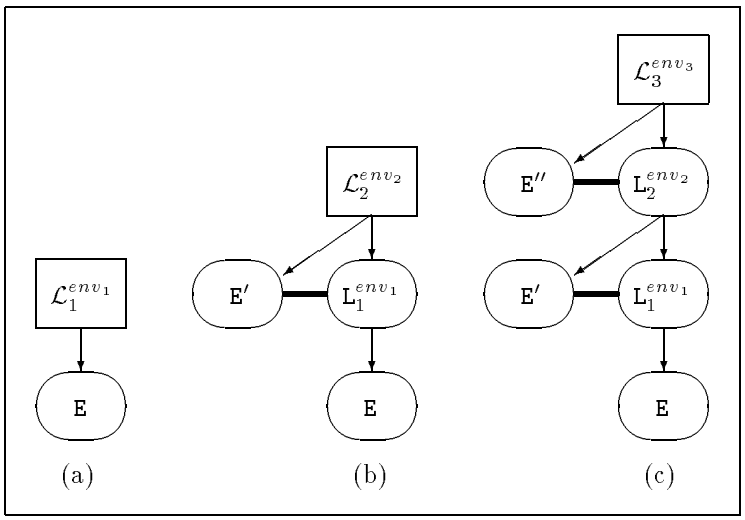

Figure 5. The exact approach

such a directly executable interpreter becomes an interpreter quite similar to $\mathtt{L}_n^{env_n}$ in shape, it differs from $\mathtt{L}_n^{env_n}$ in at least two ways. First, it is not interpreted, but directly executed (or 'compiled'). Secondly, it performs additional tasks, such as lazy creation of metalevel interpreters. Let us denote the directly executed interpreter at level n as $\mathcal{L}_n^{env_n}$, using the calligraphic font to clearly distinguish it from $\mathtt{L}_n^{env_n}$. $\mathcal{L}_n^{env_n}$ is an executable Scheme program which mimics the execution of $\mathtt{L}_n^{env_n}$, while $\mathtt{L}_n^{env_n}$ is an interpreted one which contains a magical construct, `primitive-EM`. Regardless of their resemblance, the distinction between these two, the *original* interpreter $\mathtt{L}_n^{env_n}$, and $\mathcal{L}_n^{env_n}$ which we call its *duplication*, is very important. In fact, their slight inconsistency becomes the main issue of this section.

3.1. The Exact Approach

The most straightforward approach to implementing the infinite tower is to create interpreters lazily as needed. We call this approach *exact*, since it is faithful to the operational semantics (i.e., $\mathtt{L}_n^{env_n}$) shown in Section 2.1. Figure 5 illustrates this approach: rectangles and ovals in the figure represent directly executed code and interpreted code, respectively. Initially, a user program `E` is interpreted by a directly executed interpreter $\mathcal{L}_1^{env_1}$ (Figure 5(a)). When `(exec-at-metalevel E′)` is executed in `E`, $\mathtt{L}_1^{env_1}$ as well as $\mathcal{L}_2^{env_2}$ is created, and from thereon `E′` as well as $\mathtt{L}_1^{env_1}$ is interpreted by $\mathcal{L}_2^{env_2}$ (Figure 5(b)). As a consequence, `E` will no longer be interpreted by $\mathcal{L}_1^{env_1}$ but will be doubly interpreted by $\mathcal{L}_2^{env_2}$ and $\mathtt{L}_1^{env_1}$.

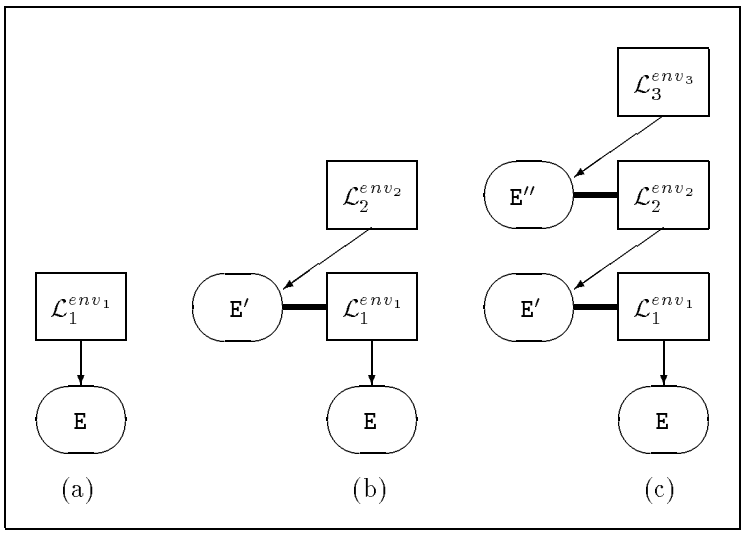

Figure 6. The approximate approach

In the case of nested reflective constructs, more interpreters are created and E will be interpreted by a stack of interpreters until it reaches an executable one (Figure 5(c)). (Usually, identical interpreters are used for $\mathcal{L}_1^{env_1}$, $\mathcal{L}_2^{env_2}$..., since they are isomorphic and are not executed simultaneously.) Note that $\mathcal{L}_n^{env_n}$ is distinct from $\mathtt{L}_n^{env_n}$. $\mathcal{L}_n^{env_n}$ is the actual executing interpreter that takes care of lazy creation of interpreters, whereas $\mathtt{L}_n^{env_n}$ is a 'conceptual' interpreter that is supposedly running, giving the semantics of the level below.

Although this approach is faithful to the operational semantics $\mathtt{L}_n^{env_n}$, it yields quite an inefficient interpreter. Once reflection is used, a complete metalevel interpreter $\mathtt{L}_1^{env_1}$ is created, even if only a small part of it is redefined. There is no way to access and use the efficient directly executed interpreter $\mathcal{L}_1^{env_1}$. This is unfortunate, because user programs do not usually change interpreters drastically, but rather make a small change and leave most other parts unchanged.

3.2. The Approximate Approach

The approximate approach, on the other hand, provides us with access to directly executed interpreters. Instead of constructing a metalevel interpreter $\mathtt{L}_n^{env_n}$, a directly executed interpreter $\mathcal{L}_n^{env_n}$ is presented to user programs as an approximation of $\mathtt{L}_n^{env_n}$. Figure 6 illustrates the scenario. Initially, a user program E is interpreted by $\mathcal{L}_1^{env_1}$ (Figure 6(a)) just the same as the exact approach. When a reflective construct in E is executed to run E′ at level 1, $\mathcal{L}_2^{env_2}$ (which is typically the same as $\mathcal{L}_1^{env_1}$ except for the environment) is created *without* constructing $\mathtt{L}_1^{env_1}$, and E′

is interpreted by $\mathcal{L}_2^{env_2}$. As a result, $\mathtt{E}'$ can access evaluator functions of the efficient interpreter $\mathcal{L}_1^{env_1}$. Furthermore, after the execution of $\mathtt{E}'$, $\mathtt{E}$ will be efficiently interpreted by $\mathcal{L}_1^{env_1}$ again, instead of doubly interpretated by $\mathtt{L}_1^{env_1}$ and $\mathcal{L}_2^{env_2}$.

When accessing $\mathcal{L}_1^{env_1}$ from $\mathtt{E}'$, we may have to convert data representation used in $\mathtt{E}'$ into the one in $\mathcal{L}_1^{env_1}$, because $\mathtt{E}'$ is an interpreted program whereas $\mathcal{L}_1^{env_1}$ is an interpreting program, and they might use different data representations. In Blond [2], this conversion is represented as the application of shifting operators $\hat{}$ and $\check{}$. Although it is possible to perform the same operations in Black every time $\mathcal{L}_1^{env_1}$ and $\mathtt{E}'$ interact, it is not desirable for efficiency reasons, especially in the presence of side-effects (cyclic data structure). As is done in the actual implementation of Blond, this conversion can be avoided by using the same data representation in both $\mathcal{L}_1^{env_1}$ and $\mathtt{E}'$.

Although the approximate approach avoids double interpretation and allows us to access directly executed interpreters, it works well only if $\mathtt{E}'$ leaves no side-effects to the metalevel interpreter. Even if user programs redefine $\mathtt{L}_1^{env_1}$ using reflection, the duplicated approximation $\mathcal{L}_1^{env_1}$ is not affected by the redefinition. Because user programs are given the illusion that the metalevel interpreter is $\mathtt{L}_n^{env_n}$, modification attempt on $\mathtt{L}_n^{env_n}$ will cause inconsistency between the actually running interpreter and the interpreter user programs expect to see. That is to say, by allowing access to directly executed interpreters, we lose redefinability of metalevel interpreters.

3.3. Duplication with Sharing (Our Approach)

One way to re-introduce redefinability is to directly resolve the inconsistency. Whenever $\mathtt{L}_n^{env_n}$ is redefined to $\mathtt{L}'^{env_n}_n$, we could make a corresponding $\mathcal{L}'^{env_n}_n$ dynamically from $\mathtt{L}'^{env_n}_n$ and $\mathcal{L}_{n+1}^{env_{n+1}}$. Although this method is interesting *per se*, it involves dynamic compilation using a partial evaluator in general, and is beyond the scope of this paper.

The approach we take here is more conservative: we use a better approximation for the tower of interpreters. The simple approximate approach described in the previous section does not allow replacement of evaluator functions, because the tower is approximated by only a single interpreter $\mathcal{L}_n^{env_n}$. By using more interpreters, we can approximate it more accurately, making side-effects 'effective'. It may seem that this leads to the original problem: inefficiency due to double (triple, etc.) interpretation. In case of using exactly two interpreters, however, we can avoid it through partial evaluation, which we will discuss in Section 4.

Figure 7 illustrates our approach before the partial evaluation is applied. Initially, a user program $\mathtt{E}$ is doubly interpreted by two interpreters, $\mathtt{L}_1^{env_1}$ and $\mathcal{L}_2^{env_2}$ (Figure 7(a)). When reflection is used, $\mathtt{L}_2^{env_2}$ and $\mathcal{L}_3^{env_3}$ are created and $\mathtt{E}'$ is interpreted by these new interpreters, possibly accessing $\mathtt{L}_1^{env_1}$ (Figure 7(b)).

The important thing to notice here is that the environment env_2 is *shared* between $\mathtt{L}_2^{env_2}$ and $\mathcal{L}_2^{env_2}$, where env_2 contains bindings of the level 1 interpreter $\mathtt{L}_1^{env_1}$. Because side-effecting operations in $\mathtt{E}'$ are interpreted in $\mathtt{L}_2^{env_2}$ as changes on env_2, the changes propagate to $\mathcal{L}_2^{env_2}$ through this sharing, correctly reflecting the user

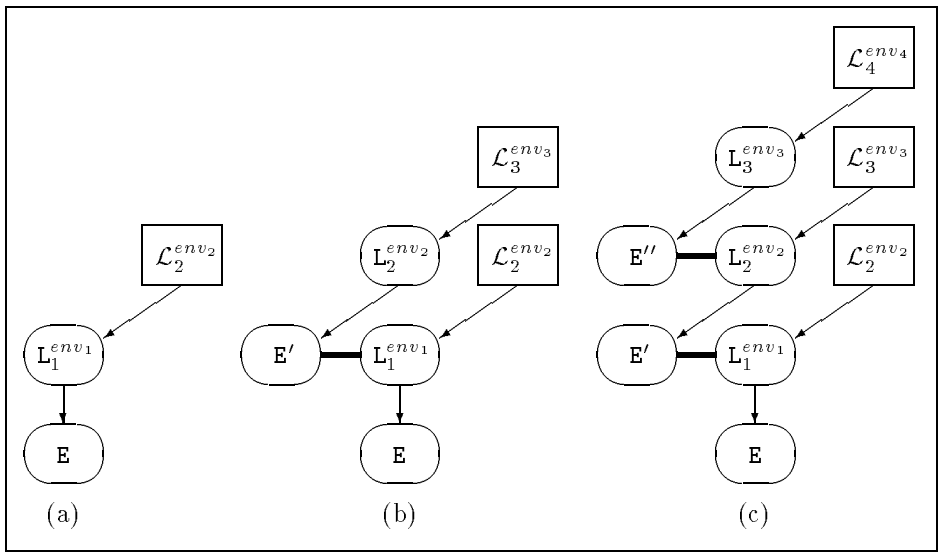

Figure 7. Our approach

modification on level 1. We show later that double interpretation by $\mathtt{L}_1^{env_1}$ and $\mathcal{L}_2^{env_2}$ can be reduced to single interpretation by partial evaluation, giving us the ability to both access the directly executed interpreters *and* to redefine them. The efficiency degradation due to user modification is minor: in the modified $\mathtt{L}_1^{env_1}$, interpretation by $\mathtt{L}_2^{env_2}$ and $\mathcal{L}_3^{env_3}$ occurs only when the modified code is executed. Other unmodified parts are directly executed as before. Considering that user modifications are usually small, we can benefit from both the efficiency of direct execution and the flexibility of redefining metalevel interpreters.

3.4. Evaluator Functions as Primitives

Although we now have the metalevel interpreter, which is redefinable through duplication with sharing, it is still an approximation of a reflective tower, since we employed only two interpreters. A natural question to ask, then, is whether or not it is sufficient. In other words, what are possible in this framework and what are not?

Redefinition of metalevel evaluator functions are certainly allowed as explained in the previous section. Every function of the metalevel interpreter is available to user programs for access or redefinition. It enables us, for example, to extend the language by adding new special forms, or to change the behavior of the language for debugging purposes. We can redefine the metalevel interpreter, as long as the language used to program the metalevel is confined to the original Scheme.

However, the framework does not support side-effects that involve interactions of more than one level. For example, side-effects to the *metametalevel* interpreter cause the following inconsistency: suppose that we want to modify the behavior of baselevel programs by going up two levels and redefining $\mathtt{L}_2^{env_2}$ so that evaluator functions in $\mathtt{L}_1^{env_1}$ are interpreted differently (Figure 7(c)). For example, we might debug `E` by tracing each function application in $\mathtt{L}_1^{env_1}$. This would be possible with 'exact' version of Black in the following way:

```
(exec-at-metalevel (exec-at-metalevel
  (let ((old-eval-application eval-application))
    (set! eval-application (lambda (exp env cont)
                             (write exp)(newline)
                             (old-eval-application exp env cont))))))
```

Unfortunately, this will not work as expected with our approach: although $\mathtt{L}_2^{env_2}$ itself would be modified through the sharing of env_3 between $\mathtt{L}_3^{env_3}$ and $\mathcal{L}_3^{env_3}$, $\mathcal{L}_2^{env_2}$ does not change, since $\mathtt{L}_2^{env_2}$ and $\mathcal{L}_2^{env_2}$ are distinct interpreters that share env_2 only. After the modification, $\mathtt{E}'$ is interpreted by the modified $\mathtt{L}_2^{env_2}$ as expected, but `E` is interpreted by $\mathtt{L}_1^{env_1}$ which is still interpreted by the original $\mathcal{L}_2^{env_2}$. Thus, changes on $\mathtt{L}_2^{env_2}$ affect only the metalevel user code $\mathtt{E}'$. The behavior of the metalevel interpreter code $\mathtt{L}_1^{env_1}$ remains unchanged, and likewise for the baselevel program `E`. In our example, no traces will be printed, since there is no user code at level 1.

Side-effects to the metametalevel interpreter could be made effective, if we used more accurate approximation, i.e., three interpreters as in Figure 8. We could duplicate $\mathtt{L}_3^{env_3}$ (not $\mathtt{L}_2^{env_2}$) to get directly executed interpreter $\mathcal{L}_3^{env_3}$ and share env_3 between two $\mathtt{L}_3^{env_3}$'s and $\mathcal{L}_3^{env_3}$, so that changes to $\mathtt{L}_2^{env_2}$ propagate through env_3 to the baselevel. Unfortunately, we cannot eliminate such triple interpretation: although partial evaluation techniques will enable us to reduce it to double interpretation, reducing it to single interpretation will amount to employing dynamic compilation. We will discuss the phenomenon in Section 4.2.

Instead, we *accept* the two level approximation, and give up side-effects to the metametalevel. This does not mean that side-effects on the metametalevel (and above) are meaningless. Although the behavior of metalevel evaluator functions is not modified by changing the metametalevel interpreter, the behavior of metalevel user code is modified. (In our example, execution of $\mathtt{E}'$ will print its trace.) If some of the metalevel evaluator functions are replaced with user-defined code, modification of the metametalevel does affect the behavior of the replaced functions, which then affect the behavior of baselevel programs. This leads us to a natural view, in which evaluator functions are *compiled* and executed as primitive functions. In Section 2.1, we stated that the evaluator functions were supposed to be interpreted by another interpreter one level above. Now, they are no longer interpreted but directly executed as primitive (or compiled) functions. Their behavior is clearly defined (as in Section 2.1), but they are executed as "black boxes." In other words, we can replace evaluator functions with user-defined functions, but cannot manipulate their internal execution.

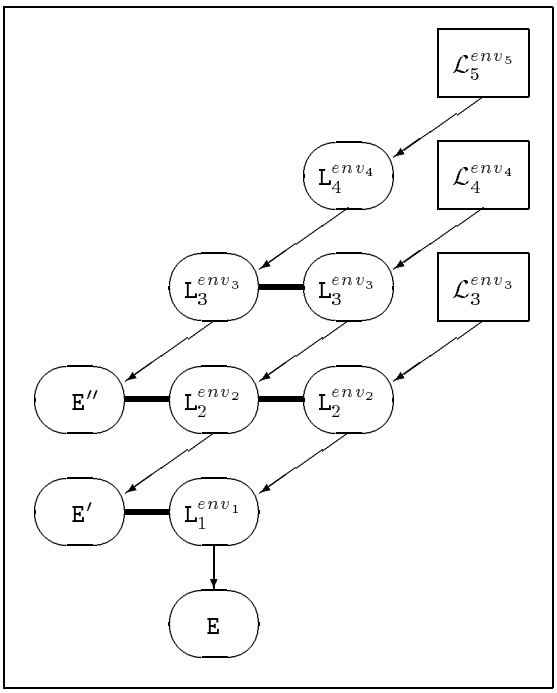

Figure 8. Using three interpreters for approximation

Note that we have already accepted such "black boxes" in the standard Scheme libraries. For example, consider `map` in Scheme: the behavior of `map` can be clearly defined as:

```
(define (map fun lst)
  (if (null? lst) '() (cons (fun (car lst)) (map fun (cdr lst)))))
```

but it is usually a compiled primitive. We can define another slightly different version of `map` and use it to replace the original `map`, but cannot change the original `map` itself.

4. Partial Evaluation

Although the infinite tower of interpreters was shown to be implementable by duplication with sharing, the resulting interpreter is quite slow due to double interpretation: a user program `E` at level $n-1$ is always interpreted by two interpreters as $\mathcal{L}_{n+1}^{env_{n+1}}(\mathtt{L}_n^{env_n}(\mathtt{E}))$. To remove this double interpretation, we employ partial evaluation techniques[8]. Knowing that $\mathcal{L}_{n+1}^{env_{n+1}}$ (other than env_{n+1}) does not change,

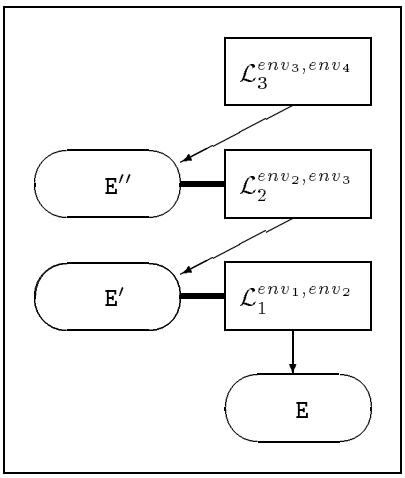

Figure 9. The tower of interpreters after the partial evaluation

we specialize $\mathcal{L}_{n+1}^{env_{n+1}}$ on the argument $\mathtt{L}_n^{env_n}$. During partial evaluation, we treat env_{n+1} as a dynamic (unknown) value, so that changes to the environment remain effective. The effect of the partial evaluation is significant: let the result of such partial evaluation be $\mathcal{L}_n^{env_n,env_{n+1}}$; as a function, it is exactly the same as $\mathcal{L}_{n+1}^{env_{n+1}}(\mathtt{L}_n^{env_n}(\cdot))$, but its structure is very similar to $\mathtt{L}_n^{env_n}$ (except that it contains accesses to env_{n+1}) and it runs almost as efficiently as $\mathcal{L}_n^{env_n}$, i.e., the directly implemented version of $\mathtt{L}_n^{env_n}$. Figure 9 shows the structure of the tower after the partial evaluation. This is in contrast to Figure 7(c), in which user programs are always interpreted by two interpreters. Note that unlike Figure 6(c), $\mathcal{L}_n^{env_n,env_{n+1}}$ allows redefinitions of evaluator functions. Operationally, Figure 9 is exactly the same as Figure 7(c).

The partial evaluation we will perform is a manual one due to several technical reasons (see Section 4.3). Instead of using an actual partial evaluator, we will partially evaluate some simple examples manually to derive portions of $\mathcal{L}_n^{env_n,env_{n+1}}$. Such examples will provide us with sufficient information to construct $\mathcal{L}_n^{env_n,env_{n+1}}$ fully, which we present in Section 5.

4.1. The Basic Idea

A partial evaluator is a function that takes a program and some (i.e., not necessarily all) of its arguments, and returns a program that is specialized to those known arguments. Given a program and some arguments, a partial evaluator executes the program as much as possible, leaving computation on unknown inputs as programs which are to be executed at runtime. The output program (called a *residual* pro-

gram) typically runs much faster than the original program for the specified inputs, as it has already performed (at partial evaluation time) the computation that uses the known arguments.

Using a partial evaluator, we can collapse $\mathcal{L}_{n+1}^{env_{n+1}}(\mathtt{L}_n^{env_n}(\cdot))$ into $\mathcal{L}_n^{env_n,env_{n+1}}$. Since $\mathcal{L}_{n+1}^{env_{n+1}}$ is basically a metacircular interpreter[4], the result of partially evaluating $\mathcal{L}_{n+1}^{env_{n+1}}(\mathtt{L}_n^{env_n}(\cdot))$ would become $\mathtt{L}_n^{env_n}$ itself, if env_{n+1} were regarded as a known value. However, because env_{n+1} is shared between $\mathcal{L}_{n+1}^{env_{n+1}}$ and $\mathtt{L}_{n+1}^{env_{n+1}}$, and may be modified implicitly by $\mathtt{L}_{n+1}^{env_{n+1}}$, we cannot partially evaluate accesses to env_{n+1}. As a result, we have to residualize such accesses: the resulting interpreter $\mathcal{L}_n^{env_n,env_{n+1}}$ has almost the same structure as $\mathtt{L}_n^{env_n}$, but has explicit accesses to env_{n+1} in places where env_{n+1} would be implicitly accessed if $\mathtt{L}_n^{env_n}$ were actually interpreted by $\mathcal{L}_{n+1}^{env_{n+1}}$.

4.2. Some Examples of Partial Evaluation

To see how accesses to env_{n+1} residualize, we now examine some simple examples. We first partially evaluate $\mathcal{L}_{n+1}^{env_{n+1}}$(`(base-eval exp env cont)`). The partial evaluation proceeds by unfolding the evaluator functions in $\mathcal{L}_{n+1}^{env_{n+1}}$. Since the argument to $\mathcal{L}_{n+1}^{env_{n+1}}$ is an application, `eval-application` (in $\mathcal{L}_{n+1}^{env_{n+1}}$) is executed to call `eval-list`. It, then, evaluates `base-eval`, `exp`, `env`, and `cont` under env_{n+1}. During the evaluation of `base-eval`, `eval-var` is called to produce the residual code `(cdr (get 'base-eval` env_{n+1}`))`[5]; this is because env_{n+1} is unknown at partial evaluation time, and we cannot obtain the value of `base-eval`. After `eval-list`, `base-apply` is called to generate code that applies `(cdr (get 'base-eval` env_{n+1}`))` to its arguments.

Assuming that the continuation of $\mathcal{L}_{n+1}^{env_{n+1}}$ is $cont_{n+1}$, the result becomes as follows:

```
(let ((proc (cdr (get 'base-eval env_{n+1}))))
  (cond ((procedure? proc)
         (cont_{n+1} (scheme-apply proc (list exp env cont))))
        ((and (pair? proc)
              (eq? (car proc) lambda-tag))
         (let ((lambda-params          (car (cdr proc)))
               (lambda-body       (car (cdr (cdr proc))))
               (lambda-env   (car (cdr (cdr (cdr proc))))))
           (eval-begin-body lambda-body
                            (extend lambda-env
                                    lambda-params (list exp env cont))
                            cont_{n+1})))
        (else
         (my-error (list 'Not 'a 'function: proc) env_{n+1} cont_{n+1}))))
```

As the result is quite similar to `base-apply` except that it first looks up env_{n+1}, let us define `meta-apply`$_n$ as follows to abstract out the environment lookup and the subsequent application:

```
(define (meta-apply_n proc-name . args)
  (let ((proc (cdr (get proc-name env_{n+1}))))
    (cond ((procedure? proc) (cont_{n+1} (scheme-apply proc args)))
          ((and (pair? proc)
                (eq? (car proc) lambda-tag))
           (let ((lambda-params          (car (cdr proc)))
                 (lambda-body       (car (cdr (cdr proc))))
                 (lambda-env   (car (cdr (cdr (cdr proc))))))
             (eval-begin-body lambda-body
                              (extend lambda-env lambda-params args)
                              cont_{n+1})))
          (else
           (my-error (list 'Not 'a 'function: proc) env_{n+1} cont_{n+1})))))
```

Then, we can rewrite the result as follows:

```
(meta-apply_n 'base-eval exp env cont)
```

After abstracting **meta-apply**$_n$ out, we can observe that the result has just the same structure as the original expression except for the presence of **meta-apply**$_n$. What does **meta-apply**$_n$ do, then? It is quite similar to **base-apply** in $\mathtt{L}_{n+1}^{env_{n+1}}$, except that it first looks up the metalevel environment env_{n+1} to obtain the value of **base-eval**. (Note that this environment access cannot be performed at partial evaluation time.) After the value of **base-eval** is obtained, it is applied normally as in **base-apply**: if the value of **base-eval** is an executable function (identified by the **procedure?** primitive), **meta-apply**$_n$ simply applies it to its arguments. If the value is a user-defined function (which means that **base-eval** had been replaced by the user), it calls $\mathcal{L}_{n+1}^{env_{n+1}}$ (via **eval-begin-body**) to interpret the user-defined **base-eval** correctly. In other words, **meta-apply**$_n$ effectively encapsulates all the tasks that are necessary at the metalevel to maintain the changeability property of the evaluator functions.

Let us see another example. By specializing $\mathcal{L}_{n+1}^{env_{n+1}}$ with respect to **eval-if**:

```
(base-eval (car (cdr exp)) env
           (lambda (pred)
             (if pred
                 (base-eval (car (cdr (cdr exp))) env cont)
                 (base-eval (car (cdr (cdr (cdr exp)))) env cont))))
```

we get:

```
(meta-apply_n 'base-eval (car (cdr exp)) env
              (lambda (pred)
                (if pred
                    (meta-apply_n 'base-eval (car (cdr (cdr exp))) env cont)
                    (meta-apply_n 'base-eval
                                  (car (cdr (cdr (cdr exp)))) env cont))))
```

Again, the structure of the result is just the same as the original definition of `eval-if` except for $\texttt{meta-apply}_n$. Since the value of other evaluator functions may be user-defined functions, the partial evaluator residualizes the code of $\mathcal{L}_{n+1}^{env_{n+1}}$ in $\texttt{meta-apply}_n$, so that they become sensitive to redefinition.

We can make several important observations on the above examples. First, we obtain $\mathcal{L}_n^{env_n, env_{n+1}}$ by just inserting $\texttt{meta-apply}_n$'s into the appropriate places. All the code that is necessary to make evaluator functions replaceable is residualized in $\texttt{meta-apply}_n$. Secondly, the resulting $\mathcal{L}_n^{env_n, env_{n+1}}$ runs far more efficiently than $\mathcal{L}_{n+1}^{env_{n+1}}(\mathtt{L}_n^{env_n}(\cdot))$: since $\texttt{meta-apply}_n$ applies evaluator functions immediately when they are not redefined, $\mathcal{L}_n^{env_n, env_{n+1}}$ runs almost as efficiently as $\mathcal{L}_n^{env_n}$ and still allows us to redefine evaluator functions. That is to say, we can view $\texttt{meta-apply}_n$'s as 'hooks' that realize redefinition of evaluator functions. In conventional reflective systems, these hooks were inserted in a rather ad-hoc manner. We have succeeded in deriving these hooks in a schematic way by duplicating and partially evaluating interpreters, giving a proper semantic account of them.

Knowing that $\texttt{meta-apply}_n$'s serve as hooks, we can now make various versions of $\mathcal{L}_n^{env_n, env_{n+1}}$ without doing any partial evaluation. By inserting $\texttt{meta-apply}_n$'s to only some specified function calls, we obtain a more efficient interpreter which contains hooks at only necessary places. In the last example, $\texttt{meta-apply}_n$'s were not inserted to primitive function calls (`car` and `cdr`), since they are usually not redefinable[6]. In Section 5, we will insert $\texttt{meta-apply}_n$'s to all calls to CPS evaluator functions, because we want our interpreter to be general-purpose and make no assumption on how the metalevel interpreter is usually changed. In practice, we insert them to only interesting parts to prevent unnecessary hooks. For example, removing all $\texttt{meta-apply}_n$'s except for the call to `eval-var` in `base-eval` provides us with an interpreter in which only `eval-var` is redefinable. All other function calls are executed efficiently without hooks. This would be sufficient if `eval-var` were the only function we were interested in[7].

Note that because of the hooks, it would be difficult to reduce triple interpretation to single interpretation. The result of partially evaluating two interpreters contains a hook at every function application. Since these hooks cannot be unfolded until runtime, further partial evaluation of it with respect to another interpreter will make no progress, residualizing almost the entire two interpreters.

4.3. Notes on Automatic Partial Evaluation

Instead of using an actual partial evaluator, we partially evaluated $\mathcal{L}_{n+1}^{env_{n+1}}(\mathtt{L}_n^{env_n}(\cdot))$ manually to obtain $\mathcal{L}_n^{env_n, env_{n+1}}$. There are mainly two reasons for this. First, because $\mathtt{L}_n^{env_n}$ contains a special construct `primitive-EM` in `eval-EM`, we have to treat it specially to correctly coordinate the levels, making fully automatic partial evaluation difficult.

Secondly, since user-defined lambda closures are represented as lists in $\mathcal{L}_{n+1}^{env_{n+1}}$, partially evaluating $\mathcal{L}_{n+1}^{env_{n+1}}$ with respect to an evaluator function in $\mathtt{L}_n^{env_n}$ will not yield a lambda closure representing it, but rather a reified data structure (a

plain list) containing its parameters, body, and captured environment. For example, suppose that we specialize $\mathcal{L}_{n+1}^{env_{n+1}}$ with respect to `eval-quote` as $\mathcal{L}_{n+1}^{env_{n+1}}$(`(lambda (exp env cont) (cont (car (cdr exp))))`). Because the first element of the expression is a `lambda`, `eval-lambda` in $\mathcal{L}_{n+1}^{env_{n+1}}$ is called to evaluate the expression, which will immediately return the following data structure as a representation of the lambda closure:

`((lambda . tag) (exp env cont) ((cont (car (cdr exp)))))` env_{n+1})

To convert this into an actual executable lambda closure, we need to know that it is always decomposed and executed (using `eval-begin-body` in `base-apply`) as a lambda closure. However, automatically extracting this knowledge and incorporating it into existing partial evaluators is difficult.

Because our current goal of using a partial evaluator is to obtain enough information for constructing $\mathcal{L}_n^{env_n,env_{n+1}}$ manually, we believe our current approach is sufficient. Once we obtain $\mathcal{L}_n^{env_n,env_{n+1}}$, we do not use a partial evaluator in Black (unless we employ dynamic compilation).

5. The Implementation of Black

We shall now present the concrete implementation of Black in Scheme. This section together with the Appendix gives the complete implementation of Black. In previous sections, we have shown that the Black interpreter can be implemented by lazily creating $\mathcal{L}_n^{env_n,env_{n+1}}$. The actual strategy we employed is to lazily create a *metacontinuation* instead of $\mathcal{L}_n^{env_n,env_{n+1}}$ itself. The metacontinuation, proposed in Brown[5][13] and later refined in Blond[2], provides us with convenient means of maintaining the information of all the levels in the tower. It also enables us to share all the evaluator functions in $\mathcal{L}_n^{env_n,env_{n+1}}$, which leads to a simpler and clearer implementation.

5.1. Metacontinuation

A metacontinuation is an infinite list of pairs of an environment and a continuation [2].

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

Intuitively, it holds all the environments and continuations of the current level and above. Let the environment and continuation of level n be env_n and $cont_n$, respectively. The baselevel metacontinuation can then be written as:

$$(env_1 \times cont_1) \times (env_2 \times cont_2) \times \cdots$$

With this structure, we can "shift-up" in the tower and obtain the metacontinuation of the level above by simply removing the first element. Likewise, we "shift-down"

in the tower and obtain the metacontinuation of the level below by consing the current environment and continuation.

The metacontinuation is an infinite list, so in practice we employ a stream (i.e., a lazy list) for its implementation. Using `cons-stream`, the initial metacontinuation can be defined as follows. (See Section 5.5 for the precise definition of `init-Mcont`.)

```
(define (init-Mcont)
  (cons-stream (list (copy init-env) init-cont)
               (init-Mcont)))
```

Shifting one level up is then implemented by replacing the current metacontinuation `Mcont` with `(tail Mcont)`, while shifting one level down is achieved by replacing `Mcont` with `(cons-stream (list env cont) Mcont)`, where `env` and `cont` represent the current level environment and continuation, respectively.

5.2. Evaluator Functions without Levelshifting

To implement the Black interpreter, we create a common interpreter shared by all levels. This interpreter is then used to generate $\mathcal{L}_n^{env_n, env_{n+1}}$ by instantiating it with the appropriate metacontinuation for level n. As a result, the type of evaluator functions becomes: $Exp \times Env \times Cont \rightarrow Mcont \rightarrow Answer$, where $Mcont$ is the domain of metacontinuations. Initially, we consider only evaluator functions that do not shift levels. Metacontinuations do not play any roles here and become implicit via η-reduction. (This is analogous to *stores* in the denotational semantics, where they become explicit only in functions with side-effects.)

Figure 10 (together with Appendix A.3) shows the evaluator functions that do not shift levels. They resemble the corresponding functions in $\mathtt{L}_n^{env_n}$ in that they are simply obtained by inserting `meta-apply`'s into the applications of `cont` and other CPS evaluator functions. In both cases, the applied functions could be user-defined lambda closures, upon which `meta-apply` invokes the task of metalevel interpretation. The code of `meta-apply`, however, differs from the one in Section 4.2 because of metacontinuations. (See Section 5.4 for more details.)

Note that Figure 10 is the *Scheme implementation* of Black, and not the interpreter observed by user programs via `exec-at-metalevel`. Instead, what is observed is the interpreter $\mathtt{L}_n^{env_n}$ shown in Section 2.1[8]. Metacontinuations are merely used to implement Black in Scheme, and are not visible from user programs, even if levelshifting occurs.

5.3. Evaluator Functions with Levelshifting

Let us now consider evaluator functions that require levelshifting. The function `eval-EM` is defined as follows using a metacontinuation:

```
; Cont = Value -> Mcont -> Answer
; functions : Exp * Env * Cont -> Mcont -> Answer
; meta-apply : Symbol or Proc * List of Values -> Mcont -> Answer
(define (base-eval exp env cont)
  (cond ((number? exp)           (meta-apply cont exp))
        ((boolean? exp)          (meta-apply cont exp))
        ((string? exp)           (meta-apply cont exp))
        ((symbol? exp)           (meta-apply 'eval-var exp env cont))
        ((eq? (car exp) 'quote)  (meta-apply 'eval-quote exp env cont))
        ((eq? (car exp) 'if)     (meta-apply 'eval-if exp env cont))
        ((eq? (car exp) 'set!)   (meta-apply 'eval-set! exp env cont))
        ((eq? (car exp) 'lambda) (meta-apply 'eval-lambda exp env cont))
        ((eq? (car exp) 'begin)  (meta-apply 'eval-begin exp env cont))
        ((eq? (car exp) 'exec-at-metalevel)
                                 (meta-apply 'eval-EM exp env cont))
        ((eq? (car exp) 'exit)   (meta-apply 'eval-exit exp env cont))
        (else (meta-apply 'eval-application exp env cont))))
(define (eval-var exp env cont)
  (let ((pair (get exp env)))
    (if (pair? pair)
        (meta-apply cont (cdr pair))
        (meta-apply 'my-error
                    (list 'eval-var: 'unbound 'variable: exp) env cont))))
(define (eval-quote exp env cont) (meta-apply cont (car (cdr exp))))
(define (eval-if exp env cont)
  (meta-apply 'base-eval (car (cdr exp)) env
              (lambda (pred)
                (if pred (meta-apply 'base-eval (car (cdr (cdr exp)))
                                     env cont)
                         (meta-apply 'base-eval (car (cdr (cdr (cdr exp))))
                                     env cont)))))
(define (eval-set! exp env cont)
  (let ((var (car (cdr exp)))
        (body (car (cdr (cdr exp)))))
    (meta-apply 'base-eval body env
                (lambda (data)
                  (let ((pair (get var env)))
                    (if (pair? pair)
                        (begin (set-value! var data env)
                               (meta-apply cont var))
                        (meta-apply 'my-error
                                    (list 'eval-set!: 'unbound 'variable var)
                                    env cont)))))))
(define lambda-tag (cons 'lambda 'tag))
(define (eval-lambda exp env cont)
  (let ((lambda-body (cdr (cdr exp)))
        (lambda-params (car (cdr exp))))
    (meta-apply cont (list lambda-tag lambda-params lambda-body env))))
```

Figure 10. Evaluator functions without levelshifting

```
; eval-EM    : Exp * Env * Cont -> Mcont -> Answer
; meta-apply : Symbol or Proc * List of Values -> Mcont -> Answer
(define (eval-EM exp env cont)
  (lambda (Mcont)
    (let ((meta-env (car (head Mcont)))
          (meta-cont (car (cdr (head Mcont))))
          (meta-Mcont (tail Mcont)))
      ((meta-apply 'base-eval (car (cdr exp))
                             meta-env
                             (lambda (ans)
                               (lambda (Mcont2)
                                 ((meta-apply cont ans)
                                  (cons-stream (head Mcont) Mcont2)))))
       meta-Mcont))))
```

Notice that `eval-EM` receives a metacontinuation `Mcont` explicitly, from which it extracts the metalevel environment, continuation, and the metacontinuation of the level above[9]. Then, the body argument of `exec-at-metalevel` is evaluated under `meta-env` and `meta-Mcont`, i.e., at the metalevel. If side-effects are used during metalevel execution, `meta-env` is modified to replace the metalevel evaluator functions in the subsequent interpretation. (Recall that `meta-env` (env_{n+1} of Section 4.2) is always looked up to determine which evaluator functions to run in the execution of baselevel programs.) After executing the body, the result is applied to `cont`, at which time the metalevel environment and continuation are pushed back into the metacontinuation, so that the computation continues in the baselevel. By switching metacontinuations, the body of `exec-at-metalevel` is correctly executed at the metalevel.

We next consider the function `base-apply`. Because `base-apply` does not involve explicit reflective operations, it might seem that levelshifting would not be necessary; but in fact, it is:

```
; base-apply : Proc * List of Values * Env * Cont -> Mcont -> Answer
(define (base-apply operator operand env cont)
  (cond ((procedure? operator)
         (cond ((eq? operator map)
                (meta-apply 'eval-map
                            (car operand) (car (cdr operand)) env cont))
               ((eq? operator scheme-apply)
                (meta-apply 'base-apply
                            (car operand) (car (cdr operand)) env cont))
               ((pair? (memq operator primitive-procedures))
                (meta-apply cont (scheme-apply operator operand)))
               (else ; evaluator functions <========================== (A)
                (lambda (Mcont)
                  ((scheme-apply operator operand)
                   (cons-stream (list (get-global-env env) cont)
                                Mcont))))))
```

```
((and (pair? operator)
      (eq? (car operator) lambda-tag))
 (let ((lambda-params        (car (cdr operator)))
       (lambda-body     (car (cdr (cdr operator))))
       (lambda-env (car (cdr (cdr (cdr operator))))))
   (meta-apply 'eval-begin-body
               lambda-body
               (extend lambda-env lambda-params operand)
               cont)))
(else (meta-apply 'my-error
                  (list 'Not 'a 'function: operator) env cont))))
```

The overall structure of `base-apply` is the same as `base-apply` in $\mathtt{L}_n^{env_n}$ (except for the insertion of `meta-apply`'s). The major difference related to levelshifting is in the `else` clause of the inner `cond` statement (A). In a standard Scheme metacircular interpreter, this `else` clause will never be selected: when `operator` is a procedure, it is either one of higher order functions (of which we handle only `map` and `scheme-apply` here) or primitive procedures. However, we have evaluator functions in Black, which are directly executed primitives. Levelshifting occurs here: the application of evaluator functions in `base-apply` requires shifting down of a level.

Consider the situation in Figure 11, in which `base-eval` at level 1 is replaced with a user-defined lambda closure. (As was before, rectangles and ovals in the figure denote directly executed functions and interpreted functions, respectively.) The replaced `base-eval` is interpreted by the level 2 interpreter, whereas other evaluator functions in level 1 are directly executed. Now, suppose that the user-defined `base-eval` at level 1 calls a default (directly executed) evaluator function, such as `eval-var`, by executing `(eval-var exp env cont)`. Then, before the application, the level 2 interpreter was running (to execute the user-defined `base-eval`), whereas after the application, the level 1 evaluator function (`eval-var`) will be running, thus requiring a downward levelshift.

Let us observe more closely how `(eval-var exp env cont)` is interpreted in level 2 to understand what `base-apply` does: the expression is first passed to level 2 `base-eval`, in which the last `else` clause is selected to call level 2 `eval-application`. Then, after it calls `eval-list` to evaluate `eval-var`, `exp`, `env`, and `cont`, the level 2 `base-apply` is called to apply `eval-var`. Since `eval-var` would not be a lambda closure but rather a level 1 evaluator function, the `else` clause (A) in `base-apply` presented above is selected. There, a metacontinuation is received explicitly and the global environment and the current continuation are placed into it. After the application, computation proceeds at a level below.

We put the global environment rather than the current environment into the metacontinuation, because only the former will be required later. The only case where environments in a metacontinuation are extracted is when going up a level in `eval-EM` or `meta-apply`. In both cases, execution starts under the global environment, not the environment captured in the closure that was applied here.

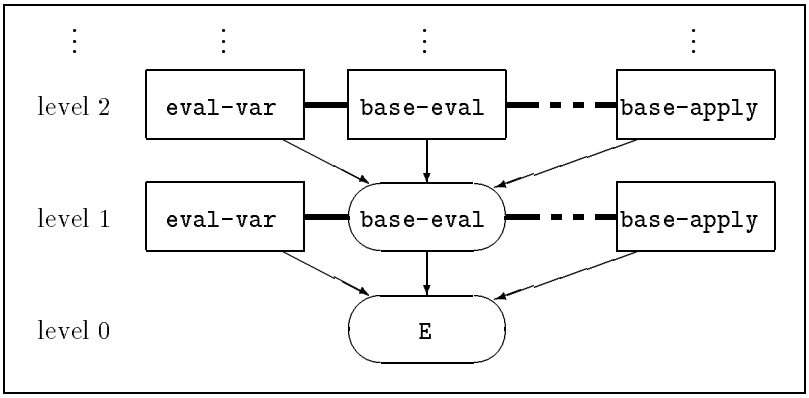

Figure 11. The tower structure when `base-eval` of level 1 is replaced

5.4. `Meta-apply`

`Meta-apply` is used to *apply* arguments to a function appropriately, calling the metalevel interpreter if necessary. Since `meta-apply` is not present in the operational semantics $\mathtt{L}_n^{env_n}$ and is instead a result of partial evaluation, it is invisible from user programs and is used exclusively by the Black language implementation in the default evaluator functions.

```
; meta-apply : Symbol or Proc * List of Values -> Mcont -> Answer
(define (meta-apply proc-name-or-cont . operand)
  (lambda (Mcont)
    (let* ((meta-env (car (head Mcont)))
           (meta-cont (car (cdr (head Mcont))))
           (meta-Mcont (tail Mcont))
           (operator (if (symbol? proc-name-or-cont)
                         (cdr (get proc-name-or-cont meta-env))
                         proc-name-or-cont)))
      (cond ((procedure? operator)
             (if (pair? (memq operator primitive-procedures))
                 ((meta-apply 'base-apply operator operand
                                          meta-env meta-cont)
                  meta-Mcont)
                 ((scheme-apply operator operand) ; evaluator functions
                  Mcont)))
            (else
             ((meta-apply 'base-apply operator operand meta-env meta-cont)
              meta-Mcont))))))
```

The code for `meta-apply` is simple, compared to the one for `base-apply`, since it delegates the task of application to the metalevel `base-apply` in most cases. After

extracting the metalevel environment, continuation, and metacontinuation from the metacontinuation (the `let*` statement), `meta-apply` looks up the `operator` in the metalevel environment. Here, we check whether or not `proc-name-or-cont` is a symbol, because `meta-apply` is used to apply `cont` as well as evaluator functions. When the `proc-name-or-cont` is not passed as a symbol, it indicates that the argument is `cont` and since `cont` is already a function (either a directly executable function or a lambda closure), we do not need to lookup the environment[10].

Given an `operator`, `meta-apply` branches according to whether or not it is an evaluator function. If the `operator` is an evaluator function, it is directly applied at the same level. On the other hand, if the `operator` turns out to be not an evaluator function, this indicates that the evaluator function has been modified by user programs. It is, then, either a primitive function, a user-defined lambda closure, or some inapplicable object (in which case an error should occur). In all cases, we delegate it to the metalevel `base-apply` to handle it appropriately at the metalevel. Note that the application of evaluator functions cannot be delegated to the metalevel `base-apply`, since that would cause infinite regression.

Finally, we move on to `my-error`:

```
(define old-env 0)
(define old-cont 0)
; my-error : Exp * Env * Cont -> Mcont -> Answer
(define (my-error exp env cont)
  (lambda (Mcont)
    (let ((meta-env (car (head Mcont)))
          (meta-cont (car (cdr (head Mcont))))
          (meta-Mcont (tail Mcont)))
      (set-value! 'old-env env meta-env)
      (set-value! 'old-cont cont meta-env)
      ((meta-apply meta-cont exp) meta-Mcont))))
```

The termination of a level is implemented by applying the metalevel continuation `meta-cont` under the metalevel metacontinuation `meta-Mcont`, throwing away the current continuation `cont`. Before that, we store the value of the current environment and continuation into the metalevel environment `meta-env` using `set-value!`. Observe how the statement `(set! 'old-env env)` appeared in `eval-exit` in $\mathtt{L}_n^{env_n}$ is simulated here. We can regard it as the result of partially evaluating $\mathcal{L}_{n+1}^{env_{n+1}}$ with respect to the `set!` statement.

5.5. Bootstrapping Functions

We now show the definition of bootstrapping functions and miscellaneous functions to complete the implementation of Black. Bootstrapping functions for the interpreter of each level are simple. Both `init-cont` and `run` are obtained by simply inserting `meta-apply`'s (Appendix A.3). `Init-env` is identical to the one in $\mathtt{L}_n^{env_n}$, and is shown in Figure 3. Environment manipulation functions and some other functions are also identical to the ones in $\mathtt{L}_n^{env_n}$. See Appendices A.1 and A.2.

Remaining functions are for the bootstrapping of the tower of interpreters. The Black system is booted by constructing a metacontinuation and executing the first continuation in it:

```
(define (black)
  (let* ((base-Mcont (init-Mcont 0 (copy init-env)))
         (env (car (head base-Mcont)))
         (cont (car (cdr (head base-Mcont))))
         (Mcont (tail base-Mcont)))
    ((cont 'start) Mcont)))
```

The metacontinuation `base-Mcont` is constructed by calling `init-Mcont`, which returns a metacontinuation of the level `level`:

```
(define (init-Mcont level env-below)
  (let ((env (copy init-env)))
    (cons-stream (list env (meta-init-cont env level env-below))
                 (init-Mcont (+ level 1) env))))
```

Here, `env-below` is an environment of the level below and is used for the sharing of environments, which is done in `meta-init-cont`:

```
(define (meta-init-cont env level env-below)
  (define-value 'init-env env-below env) ; share-env
  (display "New level loaded.") (newline)
  (lambda (result) (meta-apply 'run env level result)))
```

`Meta-init-cont` returns an initial continuation that initiates a level by running `run`. Before creating a level, it sets `init-env` in the current environment to the environment below, which contains the directly executed evaluator functions, so that they become visible to user programs.

6. Example Execution

In this section, we demonstrate some example executions of Black.

```
1 ]=> (black)
New level loaded.
New level loaded.
New level loaded.
0-0: start
0-1> (cons 1 2)
0-1: (1 . 2)
0-2>
```

The prompt shows the level and the number of iterations in the level as usual. We can see that the top three levels of the metacontinuation are initially expanded. We

need at least two, since execution of the metalevel interpreter requires a metametalevel environment to be examined whether or not the metalevel interpreter is modified. The third one can be left unexpanded, if lazy lists (`cons-stream`) are implemented to hold as promises (thunks), not only the `tail` part, but both its `head` part and `tail` part.

Let us now input the trace program given in Section 2.4.

```
0-2> (exec-at-metalevel
       (let ((old-eval base-eval))
         (set! base-eval (lambda (exp env cont)
                           (write 'trace:) (write exp) (newline)
                           (old-eval exp env cont)))))
New level loaded.
0-2: base-eval
0-3>
```

We can see a new level is created lazily here. Now, traces are displayed as expected.

```
0-3> (car (cons 1 2))
trace:(car (cons 1 2))
trace:car
trace:(cons 1 2)
trace:cons
trace:1
trace:2
0-3: 1
0-4>
```

Let us confirm that `base-eval` at level 1 is actually redefined by exiting a level.

```
0-4> (exit 'bye)
trace:(exit 'bye)
trace:'bye
1-0: bye
1-1>
```

A new level is not created here, since it has already been created above.

```
1-1> base-eval
1-1: (lambda (exp env cont)
       (write 'trace:) (write exp) (newline) (old-eval exp env cont))
1-2>
```

Because the level 2 interpreter is now running, traces are no longer displayed.

Although we could confirm that `base-eval` in level 1 has actually been redefined, we cannot observe the binding of `old-eval` in `base-eval`. To inspect an environment captured in a lambda closure, we install at level 1 two new special forms, `break` and `inspect` (Figure 12), by changing the level 2 interpreter. `Break` opens a

read-eval-print loop in the environment where the `break` was executed when its argument (if any) evaluates to *true*. Similarly, `inspect` opens a loop in the captured environment of its argument closure. They together enable us to set breakpoints in a program and navigate among environments to inspect the value of variables.

```
1-2> (exec-at-metalevel (load "break.blk"))
New level loaded.
1-2: done
1-3>
```

We use here `inspect` to see the value of `old-eval` in `base-eval`.

```
1-3> (inspect base-eval)
inspect-loop
inspect> base-eval
(lambda (exp env cont)
  (write 'trace:) (write exp) (newline) (old-eval exp env cont))
inspect> old-eval
#[compound-procedure 2 base-eval]
inspect> (exit 'good-bye)
inspect-end
1-3: good-bye
1-4>
```

To exit the `inspect` loop, we simply use `exit`.

Since the continuation of level 1 is stored in `old-cont`, we can resume the execution of level 1 by calling it.

```
1-4> (old-cont 'hello)
0-4: hello
0-5>
```

Observe that the prompt says `0-4:`. At level 0, `hello` is regarded as a return value of `(exit 'bye)`, input at the prompt `0-4>`. Because we are now at level 0 again, traces are displayed.

```
0-5> (+ 1 2)
trace:(+ 1 2)
trace:+
trace:1
trace:2
0-5: 3
0-6>
```

We cannot use `inspect` here, because it is installed only at level 1[11].

```
0-6> (inspect base-eval)
trace:(inspect base-eval)
trace:inspect
1-4: (eval-var: unbound variable: inspect)
1-5>
```

```
(define (loop prompt env ans)
  (write ans) (newline)
  (display prompt) (display "> ")
  (base-eval (read) env (lambda (ans) (loop prompt env ans))))
(define (eval-break pred env cont)
  (define (run-break-loop)
    (let ((result (loop "break" env 'break-loop)))
      (write 'break-end) (newline)
      (cont result)))
  (if (null? pred)
      (run-break-loop)
      (base-eval (car pred) env
                 (lambda (pred) (if pred (run-break-loop) (cont #f))))))
(define (eval-inspect closure env cont)
  (base-eval closure env (lambda (closure)
  (let ((lambda-env (car (cdr (cdr (cdr closure))))))
    (let ((result (loop "inspect" lambda-env 'inspect-loop)))
      (write 'inspect-end) (newline)
      (cont result))))))
(let ((original-eval-application eval-application))
  (set! eval-application
        (lambda (exp env cont)
          (cond ((eq? (car exp) 'break)
                 (eval-break (cdr exp) env cont))
                ((eq? (car exp) 'inspect)
                 (eval-inspect (car (cdr exp)) env cont))
                (else
                 (original-eval-application exp env cont))))))
```

Figure 12. New special forms: `break` and `inspect` in `break.blk`

Note that an error causes a levelshift. Errors are just another way to exit a level, but with an error message as the exit value.

7. Implementing the Delta Abstraction

The implementation framework presented in this paper is not limited to constructing Black. Here, we define a different language, which has the *delta* abstraction introduced by Blond[2] instead of `exec-at-metalevel`, and show how its interpreter can be constructed in our implementation framework.

A delta abstraction is a reflective procedure or a *reifier* that receives the current expression, environment, and continuation as its arguments, and evaluates its body at the metalevel. It differs from `exec-at-metalevel` in that it explicitly receives the baselevel information, which enables us to extend the language without modifying the metalevel interpreter. For example, since the current expression is passed in as an unevaluated form, we can define the `bound?` special form, which returns *true* if its argument is bound and *false* otherwise, as follows[12]:

```
(define bound? (delta (exp env cont)
                 (cont (pair? (get (car exp) env)))))
```

By manipulating continuations, we can also define various variants of `call/cc` using the delta abstraction[2].

7.1. The Interpreter Observed by User Programs

To build a system with delta abstractions in our framework, we first define an interpreter that is observed by user programs. It becomes almost the same as $\mathtt{L}_n^{env_n}$. Here, we obtain it by making small changes to $\mathtt{L}_n^{env_n}$.

A new clause is added to the `cond` statement in `base-eval` to handle the delta special form:

```
((eq? (car exp) 'delta)  (eval-delta exp env cont))
```

`Eval-delta` returns a list representing a delta abstraction, using a unique `cons` cell `delta-tag`, just as `eval-lambda` uses `lambda-tag` to represent a lambda abstraction.

```
(define delta-tag (cons 'delta 'tag))
(define (eval-delta exp env cont)
  (let ((delta-body (cdr (cdr exp)))
        (delta-params (car (cdr exp))))
    (cont (list delta-tag delta-params delta-body))))
```

`Eval-application` has to be able to handle delta abstractions now.

```
(define (eval-application exp env cont)
  (base-eval
    (car exp) env
    (lambda (operator)
      (if (and (pair? operator) (eq? (car operator) delta-tag))
          (apply-delta operator (cdr exp) env cont)
          (eval-list (cdr exp) env
                     (lambda (operand)
                       (base-apply operator operand env cont)))))))
```

Instead of evaluating both the function and arguments parts of `exp`, it first evaluates the function part only and branches according to it. When it is a delta abstraction, it calls `apply-delta` without evaluating the arguments part. Otherwise, it evaluates the arguments part and calls `base-apply` as before.

The function `apply-delta`, which should evaluate the body of delta abstractions at the metalevel, cannot be defined directly, since delta abstractions are levelshifting operators or *reifiers*. Instead of defining it, we only define that the application of delta abstractions is handled in `apply-delta`. Because we know that the internal execution of evaluator functions becomes invisible later, we define `apply-delta` as

a black box procedure. This is similar to the case of `eval-EM` in $\mathtt{L}_n^{env_n}$ of the original Black.

Finally, we register these newly introduced functions (and a variable) on `init-env`. We add the following bindings to `init-env`:

```
(cons 'delta-tag   delta-tag)
(cons 'eval-delta  eval-delta)
(cons 'apply-delta apply-delta)
```

7.2. Implementation

Having defined the interpreter observed by user programs, it is straightforward to obtain its implementation. In most cases, we simply insert `meta-apply`'s. The clause to be inserted in `base-eval` becomes:

```
((eq? (car exp) 'delta)  (meta-apply 'eval-delta exp env cont))
```

The new evaluator functions are:

```
(define (eval-application exp env cont)
  (meta-apply
    'base-eval (car exp) env
    (lambda (operator)
      (if (and (pair? operator) (eq? (car operator) delta-tag))
          (meta-apply 'apply-delta operator (cdr exp) env cont)
          (meta-apply 'eval-list (cdr exp) env
                      (lambda (operand)
                        (meta-apply 'base-apply operator operand
                                                    env cont)))))))
(define delta-tag (cons 'delta 'tag))
(define (eval-delta exp env cont)
  (let ((delta-body (cdr (cdr exp)))
        (delta-params (car (cdr exp))))
    (meta-apply cont (list delta-tag delta-params delta-body))))
```

Only `apply-delta` needs a metacontinuation:

```
(define (apply-delta operator operand env cont)
  (lambda (Mcont)
    (let ((meta-env (car (head Mcont)))
          (meta-cont (car (cdr (head Mcont))))
          (meta-Mcont (tail Mcont)))
      (let ((delta-params    (car (cdr operator)))
            (delta-body (car (cdr (cdr operator)))))
        ((meta-apply 'eval-begin-body delta-body
                     (extend meta-env delta-params
                             (list operand env cont))
                     meta-cont)
         meta-Mcont)))))
```

After `meta-env` and `meta-cont` are extracted from the metacontinuation, the body of the delta abstraction is evaluated under `meta-cont`, using `meta-env` extended with the baselevel information. Finally, the same bindings as the user views are registered on `init-env`.

Our implementation of the delta abstraction[13] allows side-effects to the metalevel interpreters, since `meta-apply`'s are used to call evaluator functions. This is in contrast to Blond, in which the only way to extend the language is to define a reifier at the baselevel. To disable side-effects in our framework, we just throw away all the `meta-apply`'s and take the simple approximate approach shown in Section 3.2.

8. Related Work

Since Black was originally designed to clarify various semantic aspects of Rscheme [6], it is strongly influenced by Rscheme. In both Black and Rscheme, the default evaluator functions are directly executed, and can be replaced freely by user-defined functions. The contribution of Black is in its general implementation framework. In Rscheme, replacement of evaluator functions was achieved by indirection via a table called the System Object Table (SOT). However, SOT was a rather ad-hoc addition to the metalevel interpreter, and its semantic significance or necessity was not clear. Through duplication and partial evaluation of the infinite reflective tower, we have shown that SOT can be semantically subsumed by the metalevel environment.

Jefferson and Friedman[7] implemented a finite reflective tower $\mathcal{I}_R$ by executing a metacircular interpreter on top of itself. Although $\mathcal{I}_R$ provides high flexibility as in Black and Rscheme, it suffers from extreme execution overhead, and furthermore, it implements only a finite account of the tower. By duplicating and partially evaluating their interpreter, we obtain an infinite reflective tower which is directly executable. The global environment is shared by all the levels in $\mathcal{I}_R$ because of efficiency considerations, but in Black, each level has its own global environment for independent customization.

Refci[11] is another reflective language that allows redefinition of interpreters under direct execution. Refci divides the interpreter into two parts called *prelim* and *dispatch*, and allows their access and redefinition by the user. The language could be considered as a special instance of our framework, where the user-visible interpreter consists of just prelim and dispatch. In our framework, we can divide the interpreter in an arbitrary way by inserting `meta-apply`'s into appropriate places. In particular in Black we have implemented here, all CPS evaluator functions are exposed to user programs for access and redefinition.

Metacontinuations were introduced in Brown[5][13] and later refined in Blond [2]. Both systems achieved the infinite tower of directly executed interpreters using metacontinuations as in Black. However, they do not allow redefinition of the interpreter under the assumption that the identical interpreter runs throughout all levels. This limits the changes and extensions that are possible. Black allows us to freely redefine the evaluator functions, simultaneously achieving the direct

execution of the default interpreters. In our framework, Brown and Blond can be understood as sharing nothing between the original and the duplication. Because the metalevel environment is not shared and is compiled away at partial evaluation time, inconsistency between them arises and side-effects are ignored.

3-LISP[4][12] is the pioneering work in Lisp-based reflective languages. Its implementation, however, is complex and not clearly explained. Because the global environment of 3-LISP is shared by all levels, it is impossible to replace an evaluator function at a particular level[3]. We conjecture that the 3-LISP interpreter is similar to the Black interpreter implemented without using metacontinuations.

9. Conclusion

We have shown a three-staged general framework for the implementation of reflective languages: construction of the interpreter of each level, duplication, and partial evaluation. The resulting languages have the following favorable properties:

- User programs are allowed to access and change the metalevel interpreters.
- Reflective facilities are available at every level.
- The interpreter runs efficiently via direct execution.

Although the interpreter obtained with this scheme is still an approximation of the reflective tower, it gives us a consistent and natural view of the reflective tower by regarding evaluator functions as compiled primitives. Using this scheme, we have actually implemented the reflective language Black and demonstrated several examples. The framework shown here is simple and general enough to help understanding various reflective languages. In fact, we have shown how the delta abstraction in Blond can be realized in our framework.

The major challenge we are now undertaking is the compilation of the modified metalevel interpreter using partial evaluation techniques. Since we do not employ dynamic compilation in this paper, interpreters get slower when they are drastically modified. By partially evaluating a metalevel interpreter with respect to the modified interpreter, we hope to obtain a modified interpreter that is directly executable. It would also lead to compilation of reflective programs themselves.

Acknowledgments

We are grateful to Hidehiko Masuhara for his suggestions and extensive discussions on reflection. We would also like to thank Olivier Danvy, Yuuji Ichisugi, Daniel Friedman, John Simmons II, Carolyn Talcott, Takuo Watanabe, Shigeru Chiba, and Jeff McAffer for their helpful comments. Finally, we thank anonymous referees for their precise comments and criticisms, which helped to considerably improve the paper.

Notes

1. `EM` is an abbreviation of `exec-at-metalevel`.
2. We use the term 'primitive' here, since its arguments are evaluated before application. Note that `exec-at-metalevel` is a special form and does not evaluate its body beforehand. In `eval-EM`, we cannot use `exec-at-metalevel`, because the expression to be evaluated at metalevel is *the value of* `(car (cdr exp))`, not `(car (cdr exp))` itself.
3. `Scheme-apply` used here is the `apply` of the underlying Scheme, which applies a function to its arguments. In this paper, we do not use the name `apply`, because it is confusing in the presence of `base-apply` and `meta-apply` (which appears later).
4. We say 'basically' because $\mathcal{L}_{n+1}^{env_{n+1}}$ contains additional code for lazy creation of interpreters, etc. In this section, we assume that $\mathrm{L}_n^{env_n}$ contains no levelshifting operations (such as `primitive-EM` in `eval-EM`) and ignore such code. `Eval-EM` needs special treatment and will be described in Section 5.3.
5. We ignore the case in which `base-eval` is unbound in env_{n+1}.
6. There is another reason for this: since primitive functions are not written in CPS, we cannot use $\texttt{meta-apply}_n$ directly, but need to inline them in the code of `eval-if`. If primitive functions in the user observable interpreter were written in CPS, we could obtain the result by inserting $\texttt{meta-apply}_n$'s. The same argument applies to other evaluator functions written in direct style, which include environment manipulation functions, such as `get` and `extend`, shown in Appendix A.2.
7. Suppose we want to make a trace of variable accesses.
8. Or more correctly, the *compiled* version of it, whose internal behavior cannot be modified by changing its metalevel interpreter.
9. Do not confuse the three similarly-named variables: `Mcont`, `meta-cont`, and `meta-Mcont`. `Mcont` is a metacontinuation of the baselevel which holds the information about the levels above the baselevel, `meta-cont` is a continuation of the metalevel interpreter, and `meta-Mcont` is a metacontinuation of the metalevel which holds the information about the levels above the metalevel.
10. Notice that when `proc-name-or-cont` is a symbol, we still ignore the case where it is unbound in `meta-env` as was done in Section4.2. This is justified because it can *never* be unbound: we use $\texttt{meta-apply}$'s exclusively for the application of evaluator functions which are registered in `init-env`.
11. Even if `eval-inspect` were installed at level 0, an error would occur for this expression, because `eval-inspect` in Figure 12 assumes that its argument is always a lambda closure.
12. Assuming that continuations are 'pushy'[2].
13. The implementation of delta abstractions presented in this section is also obtainable via ftp.

References

1. Abelson, H., and G. J. Sussman with J. Sussman *Structure and Interpretation of Computer Programs*, Cambridge: MIT Press (1985).
2. 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).
3. des Rivières, J. private communication, (September 1993).
4. des Rivières, J., and B. C. Smith "The Implementation of Procedurally Reflective Languages," *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 331–347 (August 1984).

5. 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).
6. 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).
7. Jefferson, S., and D. P. Friedman "A Simple Reflective Interpreter," *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, pp. 24–35 (November 1992), the extended version also appears in this volume.
8. Jones, N. D., C. K. Gomard, and P Sestoft *Partial Evaluation and Automatic Program Generation*, New York: Prentice-Hall (1993).
9. Kiczales G. "Towards a New Model of Abstraction in Software Engineering," *Proceedings of the International Workshop on Reflection and Meta-Level Architecture*, pp. 1–11 (November 1992).
10. Rees, J., and W. Clinger *Revised*^{3} *Report on the Algorithmic Language Scheme*, SIGPLAN NOTICE, Vol. 21, No. 12, (December 1986).
11. Simmons II, J. W., S. Jefferson, and D. P. Friedman "Language Extension Via First-class Interpreters," Indiana University Technical Report No. 362, (September 1992).
12. 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).
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

The Appendix contains the code that did not appear in the main body of the paper. The code in Section 5 together with this Appendix comprise the complete Scheme implementation of Black, which can also be obtained via `ftp://camille.is.s.u-tokyo.ac.jp`(133.11.12.1)`/pub/black/black.tar.Z`. To start Black, load all the code in Section 5 (other than Section 5.1) and Appendix and invoke `(black)` after defining `scheme-apply`:

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

Note that some primitive functions (A.1) and environment manipulation functions (A.2) are used both in the actual implementation and a part of the user observable interpreter $\mathtt{L}_n^{env_n}$.

A.1. Primitive Functions

Two primitive functions `procedure?` and `write` must be redefined to treat lambda closures appropriately.

```
(define (primitive-procedure? . operand)
  (let ((arg (car operand)))
    (or (procedure? arg)
        (and (pair? arg)
             (eq? (car arg) lambda-tag)))))
(define (primitive-write arg)
  (if (and (pair? arg)
           (eq? (car arg) lambda-tag))
      (let ((lambda-params        (car (cdr arg)))
            (lambda-body     (car (cdr (cdr arg))))
            (lambda-env (car (cdr (cdr (cdr arg))))))
        (write (cons 'lambda (cons lambda-params lambda-body))))
      (write arg)))
(define primitive-procedures
  (list car cdr cons list pair? null? not eq? eqv? equal? set-car! set-cdr!
        append newline read primitive-write + - * / = > < quotient remainder
        number? symbol? boolean? string? memq length assq primitive-procedure?
        map scheme-apply
        make-pairs extend get set-value! define-value copy get-global-env))
```

A.2. Environment Manipulation Functions

Environments are represented as a list of frames (association lists). Destructive operations on lists (`set-car!` and `set-cdr!`) are used to realize `define` and `set!`.

```
(define empty-env '(()))
(define (make-pairs params args)
  (cond ((null? params) '())
        ((symbol? params) (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)
      '()
      (let ((pair (assq var (car env))))
        (if (pair? pair)
            pair
            (get var (cdr env))))))
(define (set-value! var value env)
  (let ((pair (get var env)))
    (if (pair? pair)
        (set-cdr! pair value)
        #f)))
(define (define-value var value env)
  (let ((pair (assq var (car env))))
    (if (pair? pair)
        (set-cdr! pair value)
        (set-car! env (cons (cons var value) (car env))))))
(define (copy env)
  (define (copy-local env)
    (if (null? env)
        '()
        (cons (cons (car (car env))
                    (cdr (car env)))
              (copy-local (cdr env)))))
  (if (null? env)
      '()
      (cons (copy-local (car env))
            (copy (cdr env)))))
(define (get-global-env env)
  (define (get-global-env-local env)
    (if (null? (cdr env))
        env
        (get-global-env-local (cdr env))))
  (if (null? env)
      env
      (get-global-env-local env)))
```

A.3. Evaluator Functions without Levelshifting

These functions are obtained by simply inserting `meta-apply`'s into the corresponding functions in $\mathtt{L}_n^{env_n}$.

```
; evaluator functions : Exp * Env * Cont -> Mcont -> Answer
(define (eval-begin exp env cont)
  (meta-apply 'eval-begin-body (cdr exp) env cont))
(define (eval-begin-body body env cont)
  (define (eval-begin-local body)
    (if (null? (cdr body))
        (meta-apply 'base-eval (car body) env cont)
        (meta-apply 'base-eval (car body) env
                    (lambda (x) (eval-begin-local (cdr body))))))
  (if (null? body)
      (meta-apply 'my-error '(eval-begin-body: null body) env cont)
      (eval-begin-local body)))
(define (eval-exit exp env cont)
  (meta-apply 'base-eval (car (cdr exp)) env
              (lambda (x) (meta-apply 'my-error x env cont))))
(define (eval-list exp env cont)
  (if (null? exp)
      (meta-apply cont '())
      (meta-apply 'base-eval (car exp) env
                  (lambda (val1)
                    (meta-apply 'eval-list (cdr exp) env
                                (lambda (val2)
                                  (meta-apply cont (cons val1 val2))))))))
(define (eval-application exp env cont)
  (meta-apply 'eval-list exp env
              (lambda (l) (meta-apply 'base-apply (car l) (cdr l) env cont))))
(define (eval-map fun lst env cont)
  (if (null? lst)
      (meta-apply cont '())
      (meta-apply 'base-apply fun (list (car lst)) env
                  (lambda (x) (meta-apply 'eval-map fun (cdr lst) env
                                          (lambda (y) (meta-apply cont (cons x y))))))))
; init-cont : Env * Number * Number * Cont -> Mcont -> Answer
(define (init-cont env level turn cont)
  (meta-apply cont
    (lambda (answer)
      (write level) (write '-) (write turn) (display ": ")
      (primitive-write answer) (newline)
      (write level) (write '-) (write (+ turn 1)) (display "> ")
      (meta-apply 'base-eval (read) env
                  (lambda (ans)
                    (meta-apply 'init-cont env level (+ turn 1)
                                (lambda (cont) (meta-apply cont ans))))))))
; run : Env * Number * Value -> Mcont -> Answer
(define (run env level answer)
  (meta-apply 'init-cont env level 0
              (lambda (cont) (meta-apply cont answer))))
```