The Reflective Language Black
自己反映言語 Black

Kenichi Asai

December, 1996

Doctoral Dissertation

Department of Information Science
Graduate School of Science
University of Tokyo

To my grandfather

Abstract

Reflective programming languages provide high flexibility and extensibility by allowing user programs to access and change their language implementation (or semantics) from within themselves. Reflective languages are useful not only for extending language semantics, but also for controlling dynamic behavior of programs. 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 the high flexibility and extensibility of reflective languages, however, their implementation has been considered to be complex. In this thesis, we clarify the structure of reflective languages, show a general construction method of reflective interpreters, and discuss a compilation framework based on partial evaluation.

The thesis is divided into three parts. In the first part of the thesis, we investigate a general implementation framework for reflective languages which satisfy the following favorable properties: (1) user programs are allowed to access and change (parts of) the interpreter defining the language semantics from within the same language framework in an arbitrary way, (2) reflective facilities are available at every level (with minor approximation), 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. Based on this framework, we implement a reflective language called *Black*, a reflective extension of the functional language Scheme.

Although the Black interpreter runs efficiently at first, it becomes considerably inefficient as the modification on the interpreter becomes substantial. For more efficient execution, we introduce compilation based on partial evaluation. In the second part of the thesis, we concentrate on the construction of a partial evaluator which is powerful enough to serve as a compiler for reflective languages. Specifically, we build a partial evaluator for functional languages which can correctly handle side-effecting operations and pointer equality. These facilities are important in compiling Black programs, since they play an essential role in the Black interpreter. We employ a side-effect analysis to identify mutable data structures and use *preactions* to avoid code duplication and code elimination without disturbing the order of execution. Apart from side-effecting operations, our partial evaluator reduces more expressions than the previous one thanks to the proper treatment of pointer equality. It can now accept almost all the features of Scheme.

In the last part of the thesis, we actually apply the partial evaluation technique to compile

various programs in Black. This can be also regarded as a non-trivial application of partial evaluation. By specializing the current interpreter with respect to a program, it is compiled under the current language semantics. Furthermore, a user program is compiled under a modified language semantics by first compiling the modified interpreter and then specializing it further with respect to the user program.

Acknowledgements

First of all, I would like to thank Akinori Yonezawa. He has initiated my research work when I was a student and supported me for more than eight years after that. Without his comments, suggestions, and advices, I had even no chance to enter this interesting field of research. He also supported me for going abroad, which helped me to think in a wider perspective.

I am also grateful to Hiroshi Imai, who has always encouraged me in building my research careers. Although our research interest is somewhat different, I learned a lot from his aggressive attitude toward research work, which I believe is very important for my future research activities.

In the earlier stage of this thesis work, I received many valuable suggestions from Satoshi Matsuoka. He provided me with many comments and motivating examples, which formed the driving force of my work.

Thanks also go to Hidehiko Masuhara. Whenever I asked him for discussions, he has carefully and patiently listened to my vague ideas and clarified them from the various points of view.

I would also like to thank Olivier Danvy, who always gives me very precise and critical comments on my research. His words, "keep everything simple" formed my basic standpoint.

I received many suggestive comments from Masami Hagiya, Takashi Masuda, Masato Takeichi, Takuo Watanabe, and Naoki Kobayashi while finishing writing this thesis.

Various discussions in Yonezawa Laboratory and Imai Laboratory were helpful to me both for the research itself and the way to drive research. Especially, discussions with Naoki Kobayashi, Kenjiro Taura, and Haruo Hosoya were most helpful.

Finally, I would like to thank my parents, my grandfather, my two sisters, and my wife for their constant supports in my daily life.

Contents

1 Introduction **1**

2 Overview of Black **6**
2.1 The Metalevel Interpreters Observed by User Programs 6
2.2 The Structure of the Reflective Tower 11
2.3 Notes on Metacircularity 12
2.4 An Example Program 12

3 Implementation Framework **14**
3.1 Duplication 14
3.1.1 The Exact Approach 14
3.1.2 The Approximate Approach 15
3.1.3 Duplication with Sharing (Our Approach) 16
3.1.4 Evaluator Functions as Primitives 17
3.2 Partial Evaluation 20
3.2.1 The Basic Idea 20
3.2.2 Some Examples of Partial Evaluation 21
3.2.3 Notes on Automatic Partial Evaluation 23

4 Implementation of Black **25**
4.1 Metacontinuation 25
4.2 Evaluator Functions without Levelshifting 26
4.3 Evaluator Functions with Levelshifting 26
4.4 `Meta-apply` 30
4.5 Bootstrapping Functions 31
4.6 Example Execution 32
4.7 Implementing the Delta Abstraction 36
4.7.1 The Interpreter Observed by User Programs 36
4.7.2 Implementation 37

5 Partial Evaluation **39**
5.1 Introduction to Partial Evaluation 40
5.2 The Core Language 44
5.3 Side-effect Analysis 45

5.3.1 Notation . . . 45
5.3.2 The Abstract Interpretation . . . 46
5.3.3 Collecting Mutable Cells . . . 47
5.4 Partial Evaluation . . . 47
5.4.1 Symbolic Value and Preaction . . . 47
5.4.2 Basic Strategies . . . 48
5.4.3 The Partial Evaluator . . . 49
5.4.4 Codification . . . 51
5.5 Correctness of the Partial Evaluator . . . 52
5.6 Extensions . . . 52

6 Compilation Framework **54**
6.1 Expression to be Partially Evaluated . . . 55
6.2 Integrating Partial Evaluator into Black . . . 57
6.2.1 Source Code of Evaluator Functions . . . 57
6.2.2 Sharing Environments between Partial Evaluator and Black . . . 58
6.3 Binding-Time Improvement . . . 59

7 Examples of Compilation **60**
7.1 Effect of Primitives . . . 60
7.2 Residualizing Calls to `base-apply` . . . 62
7.3 Effect of Error-Handling . . . 64
7.4 Extending the Language . . . 66
7.5 Lazy Evaluation . . . 69

8 Related Work **72**
8.1 Black Interpreter . . . 72
8.2 Partial Evaluation . . . 73
8.2.1 Side-effect Analysis . . . 73
8.2.2 Partial Evaluation of Side-effecting Operations . . . 73
8.2.3 Preactions . . . 74
8.3 Compilation of Reflective Languages . . . 75

9 Conclusion and Future Work **76**

A Miscellaneous Code **82**
A.1 Primitive Functions . . . 82
A.2 Environment Manipulation Functions . . . 83
A.3 Evaluator Functions without Levelshifting . . . 84

B Black Program for Lazy Evaluation **86**

C Correctness of the Partial Evaluator **88**
C.1 Store Semantics $\mathcal{E}$ 88
C.2 The Partial Evaluator $\mathcal{PE}$ 90
C.3 Semantics of PSval $\mathcal{E}_{PS}$ and the Correctness of $\mathcal{PD}$ 90
C.4 Correctness of the Partial Evaluator $\mathcal{PE}$ 95

List of Figures

2.1 Standard functions observed by user programs 7
2.2 Levelshifting functions observed by user programs 8
2.3 Bootstrapping functions observed by user programs 9
2.4 The tower structure of Black 11

3.1 The exact approach 15
3.2 The approximate approach 16
3.3 Our approach 17
3.4 Using three interpreters for approximation 19
3.5 The tower of interpreters after the partial evaluation 20

4.1 Evaluator functions without levelshifting 27
4.2 The tower structure when `base-eval` of level 1 is replaced 29
4.3 New special forms: `break` and `inspect` in `break.blk` 34

5.1 An Example Program 43
5.2 Abstract Interpretation 46
5.3 Symbolic Value and Preaction 47
5.4 Partial Evaluator 50
5.5 Codification 52

6.1 Overall scenario of the compilation 56

7.1 The result of compiling `(lambda (x) (+ x 1))` 61
7.2 The result of compiling the tracing `base-eval` 63
7.3 Black program to introduce `case` special form (`case.blk`) 67

C.1 Standard Semantics of the Core Language 89
C.2 Modified Partial Evaluator (Part 1) 91
C.2 Modified Partial Evaluator (Part 2) 92
C.3 Semantics of PSval $\mathcal{E}_{PS}$ 93

List of Tables

7.1 Execution and compilation time for `(fib 17)` 69
7.2 Execution and compilation time for `(fib 17)` under lazy evaluation 70

Chapter 1

Introduction

When new concepts or technologies for programming languages are developed, they are implemented in a particular language to prove their effectiveness. For example, imagine that we developed a lazy evaluation mechanism in a hypothetic world where only applicative order evaluation was known. To show the power of the new strategy, we implement it in an existing language and demonstrate how it widens the expressive power of the previous language. This process is repeated when another technique, say memoization, is devised. We again extend the language to realize this feature.

The process of extending the existing implementation (and thereby modifying its semantics) is sometimes quite tedious, because the implementation is usually given in lower-level languages. To change it, we have to fully understand the low-level implementation (either an interpreter or a compiler) and alter it carefully. When new techniques are concerned with a higher-level language only and can be described in terms of the higher-level language, we would rather want to use the same higher-level language with high-level abstractions to ease the effort of implementation.

A macro facility somewhat solves the problem. Consider, for example, that we want to introduce streams to a functional language Scheme[17]. Given three macro operations[1]: `define-macro`, ‘‘’ (`quassiquote`), and ‘`,`’ (`unquote`), we can introduce streams without changing the underlying Scheme implementation as follows:

```
(define head car)
(define (tail x) (force (cdr x)))
(define-macro (cons-stream a b) `(cons ,a (delay ,b)))
```

The problem here is that macros are for the local text substitution only. Using macros, it would be difficult to introduce new concepts that modify the global aspects of the language, such as an evaluation strategy. What we want is a more general mechanism in which *any* modification is possible in the same language framework. For this purpose, we introduce *reflection*[45].

[1] A more elaborate macro system is provided for the standard Scheme[17]. The operators presented here are just for illustration, although some Scheme implementations, such as Gambit Scheme, support them.

Reflective languages Reflection is the ability to reason about itself. A programming language is called reflective, when it can reason about its own computational process, or in other words, the semantics of the language. Although showing the internal computational process to users appears to be dangerous for the execution of programs, we are already familiar with it in a restricted way. Consider we are debugging a program using a debugger. When an error occurs, we peek a value of a variable, change it, and continue execution. All these features are concerned with the internal computational process. In reflective languages, this internal information is opened not only for the debugging purpose, but for more general usage to refer or change the current status of the computation.

To what extent we can reason about computational process depends on the design of the reflective language we are using. The wider the internal information is displayed, the more power we obtain on the computational process. In the previous debugger example, allowed operations were only the reference and modification of variables. Although we can slightly modify the subsequent behavior by changing the value of variables, we cannot, for example, introduce new language constructs, because information on variables is the only one presented for user accesses.

Our design choice Among various levels of reflective ability, we are interested in the most powerful case where *everything is displayed*, so that we can introduce arbitrary new concepts into the language. To achieve this, we provide users the entire interpreter in the same higher-level language. User programs are regarded as being interpreted by this metacircular interpreter. To change the language semantics, we first exit the current level to the level of the interpreter. There, we find the definition of the interpreter (or the language semantics itself) written in the same language. By redefining (parts of) it, we can arbitrarily change the semantics of the language.

This design choice introduces two difficult problems: efficiency and infinite regression. Since we employed the metacircular interpreter, the performance of user programs considerably degrades if naively implemented. The question is whether we can avoid the interpretation overhead without losing the ability to change the language semantics. An easy way to avoid this inefficiency would be to throw away the metacircular interpreter. Rather than employing excessive interpretation, we survey existing programs, collect all the modification that is actually useful, allow only these changes, and build a specific language for it. A debugger is an instance of this approach. We do not take this approach, however, because it restricts the possible modification on the semantics. Rather, we take the extreme case, where *any* modification is possible. We pursue how far we can go without losing this ability *and* efficiency. When we fully understand this approach, the former approach could be understood as the special case of ours where possible changes are restricted. In fact, we will see in this thesis that we can actually construct such a language.

Another problem is infinite regression. Once we employ a metacircular interpreter, a question arises: who interprets it? If we introduce another metacircular interpreter to interpret it, we need infinite number of metacircular interpreters, resulting in infinite regression. An easy way to solve this would be to use only one metacircular interpreter, and execute it directly in machine code without using extra metacircular interpreter. However, there is a

good reason to have more than one interpreters: we might want to change the semantics of the interpreter-level language to modify the user-level semantics, i.e., multi-level reflection. Rather than restricting reflective facilities to a single level, we take again the extreme case. We pursue how far we can go without losing the ability to use reflection at an arbitrary level. The specific case where only finite number of interpreters exist could be understood as the special case of our approach.

Unfortunately, we will see that the full multi-level reflection is not practical because of the efficiency consideration. The faithful implementation of the multi-level reflection requires inefficient double (triple, etc.) interpretation. Two ways to avoid this multiple interpretation would be to introduce some kind of approximation or to employ compilation using partial evaluation. In this thesis, we will consider them both. First, we will employ a two-level approximation, which allows us to use a single-level full reflection. This approximation turns out to present us a natural view of reflective languages and is interesting in its own right. Then, we will report primary results of compilation using partial evaluation later in the thesis.

With this design choice, we will construct a language called *Black*. Black is a Scheme based reflective language and has the following favorable properties:

1. User programs are allowed to access and change (parts of) the metacircular interpreters. This enables us to modify the language semantics in an arbitrary way.

2. Reflective facilities are uniformly available at every level (with minor approximation). Hence, there exists an infinite tower of metacircular interpreters, conceptually.

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

The resulting system is a reasonably efficient Scheme interpreter with the ability to extend and modify its semantics from within Scheme in an arbitrary way. This is in contrast to the conventional Lisp-based reflective languages, where only some of the above properties are fulfilled. For example, Brown[27, 48] and Blond[21] have an infinite tower of interpreters but do not allow their modification. $\mathcal{I}_R$[31] does allow changes of interpreters but is extremely inefficient due to multiple interpretation.

Compilation of Black Although the Black interpreter runs efficiently at first, it becomes considerably inefficient as the modification on the interpreter becomes substantial. We will also see that interpreter-based implementation is inherent. For more efficient execution, we introduce compilation using partial evaluation[32]. It is well known that a partial evaluator can compile a program when an interpreter defining the language semantics is given[28]. Since the language semantics changes in reflective languages, compilers based on a specific language semantics are not permissible. Using partial evaluation, we perform compilation according to the current language semantics.

To apply partial evaluation to Black, we first extend the existing partial evaluation framework for functional languages to cope with side-effecting operations and pointer equality. These facilities are important in compiling Black programs, since they play an essential role in the Black interpreter. We employ a side-effect analysis to identify mutable data structures

and use *preactions*[34] to avoid code duplication and code elimination without disturbing the order of execution. Apart from side-effecting operations, our partial evaluator reduces more expressions than the previous one thanks to the proper treatment of pointer equality. It can now accept almost all the features of Scheme.

Using this partial evaluator, the compilation of user programs is performed by integrating the Black system with the partial evaluator. This can be also regarded as a non-trivial application of partial evaluation. By specializing the current interpreter with respect to a program, it is compiled under the current language semantics. Furthermore, a user program is compiled under a modified language semantics by first compiling the modified interpreter and then specializing it further with respect to the user program.

Contribution The contribution of this thesis is summerized as follows:

- We present a new general framework for the implementation of reasonably efficient reflective languages that allow *arbitrary* (single-level) reflection on the language semantics at *any* level.
- We extend the existing partial evaluation framework for functional languages to cope with side-effects and pointer equality, which reduces more expressions than the previous partial evaluators.
- We successfully apply the partial evaluation technique to compile programs under modified semantics and demonstrate its effectiveness.

Outline of the thesis Corresponding to the above three contributions, the thesis can be divided into three parts. The first part, which consists of Chapters 2, 3, and 4, constructs the reflective language Black. In Chapter 2, the overview of Black is presented. We show the structure of the metacircular interpreter, introduce basic terminology, as well as an example program of Black. In Chapter 3, the general framework for implementing such reflective languages is presented. The two key concepts here are 'duplication' and "partial evaluation." Based on this framework, the complete implementation of Black in Scheme is given in Chapter 4. We also implement the *delta* abstraction introduced in Blond[21] using our framework to show its generality. This part has been published in [6] and [8].

Chapter 5 comprises the second part of the thesis. The main topic here is the construction of a partial evaluator for functional languages which can correctly handle side-effecting operations. We employ a side-effect analysis to identify mutable data structures and use *preactions*[34] to avoid code duplication and code elimination without disturbing the order of execution. Apart from side-effecting operations, our partial evaluator reduces more expressions than the previous one. It can now accept almost all the features of Scheme. This part is independent of the other two parts, and can be read independently. This part has been published in [4] and [5].

Chapters 6 and 7 belong to the third part, where the partial evaluation technique is applied to compile programs under modified semantics. After describing the overall scenario of compilation, we discuss how to intergate a partial evaluator to the Black interpreter in Chapter 6. Various examples of compilation are demonstrated in Chapter 7. This can also

be regarded as a non-trivial application of partial evaluation. The main idea of this part has been published in [7] and [3].

The related work is discussed in Chapter 8 and the thesis concludes in Chapter 9. We assume that readers are somewhat familiar with continuation passing style (CPS) metacircular interpreter, partial evaluation, and the abstract interpretation.

Chapter 2

Overview of Black

This chapter introduces the reflective language Black we are going to build in this thesis. Black is a Scheme-based reflective language. The only difference between Scheme and Black is the special reflective construct `(exec-at-metalevel E)`. It evaluates the expression `E` at one level above, i.e., at the interpreter-level. Let us define *levels* here. User programs are executed at level 0 or *baselevel*. The metacircular interpreter defining the baselevel language is running at level 1 or *metalevel*. The construct `(exec-at-metalevel E)` enables us to execute `E` at metalevel. Above the metalevel is the level 2 or *metametalevel*. Conceptually, there are infinite number of levels. The interpreter at level n defines the semantics of the level $n-1$ language. By nesting `exec-at-metalevel`, we can access and change the interpreter of any level. Black is a special Scheme in which programs can be executed at any level.

In the following, we will first define the metacircular interpreters that determine the semantics of each level. This step is important, since without knowing how the interpreters are defined, there is no way to extend or redefine them. Then, we show the infinite tower structure of Black, referred to as the "reflective tower." We also demonstrate an example program that modifies the interpreter, so that readers can get the flavor of possible interactions in Black. More examples are shown in Section 4.6.

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[1] (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 Chapter 4) 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 2.1, 2.2, 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`

```
(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 2.1: 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 2.1: 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))
               ((primitive? operator)
                (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 2.2: 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? primitive?)
  (cons 'primitive-procedures   primitive-procedures)
)))
(define-value 'init-env (copy init-env) init-env)
```

Figure 2.3: Bootstrapping functions observed by user programs

in Scheme, and other standard functions are defined in Figure 2.1. 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 2.2 shows functions that potentially cause a levelshift, and their related variables. The function `eval-EM`[1] is called from `base-eval` when the expression to be evaluated 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 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?`, 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 2.1. 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 2.3 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

[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 thesis, we do not use the name `apply`, because it is confusing in the presence of `base-apply` and `meta-apply` (which appears later).

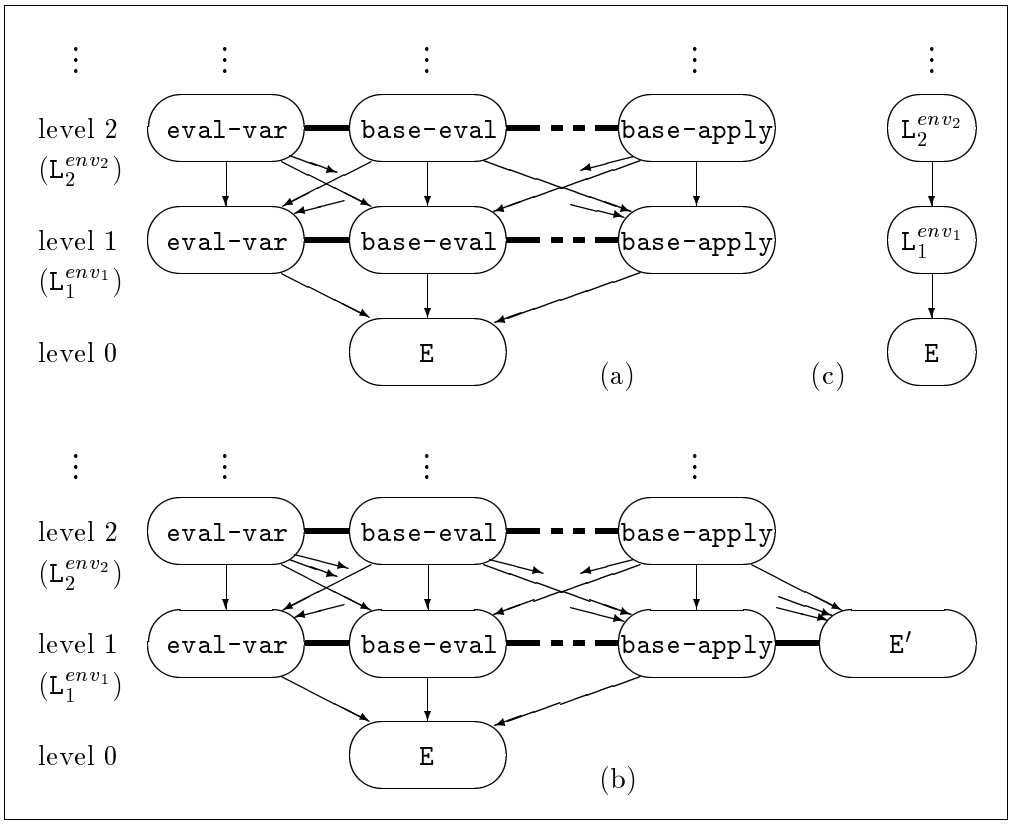

Figure 2.4: The tower structure of Black

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 2.4(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 2.4(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 2.4(c).

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. More examples are demonstrated in Section 4.6.

Chapter 3

Implementation Framework

The tower of interpreters presented in the previous chapter is unimplementable as shown, since every interpreter needs to be interpreted by the interpreter one level above, causing infinite regression. In this chapter, we consider how we can implement it in a reasonably efficient way without losing the extensibility. The two key concepts used here are 'duplication' and "partial evaluation." Duplication makes it possible to implement it in finite resources, while partial evaluation solves the efficiency problem.

3.1 Duplication

In this section, we consider how to implement the infinite reflective tower with finite resources. The basic idea is to cut the unnecessary infinite regression, and run an interpreter as executable code instead of using the interpreter one level above. Although 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.

3.1.1 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 3.1 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 3.1(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 3.1(b)). As a consequence, `E` will no longer be

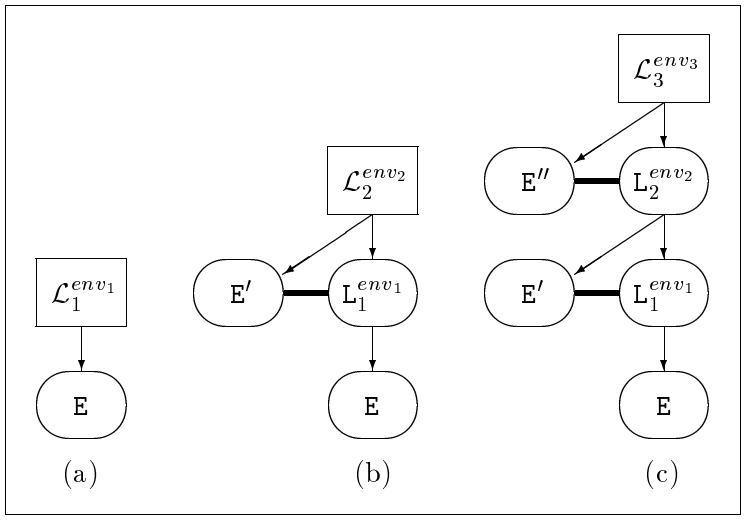

Figure 3.1: The exact approach

interpreted by $\mathcal{L}_1^{env_1}$ but will be doubly interpreted by $\mathcal{L}_2^{env_2}$ and $\mathtt{L}_1^{env_1}$. 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 3.1(c)). (Usually, identical interpreters are used for $\mathcal{L}_1^{env_1}$, $\mathcal{L}_2^{env_2}\ldots$, 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.

3.1.2 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 3.2 illustrates the scenario. Initially, a user program E is interpreted by $\mathcal{L}_1^{env_1}$ (Figure 3.2(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, E′ can access evaluator functions of the efficient interpreter $\mathcal{L}_1^{env_1}$. Furthermore, after the execution of E′, 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 E′, we may have to convert data representation used in E′ into the one in $\mathcal{L}_1^{env_1}$, because E′ is an interpreted program whereas $\mathcal{L}_1^{env_1}$ is an interpreting

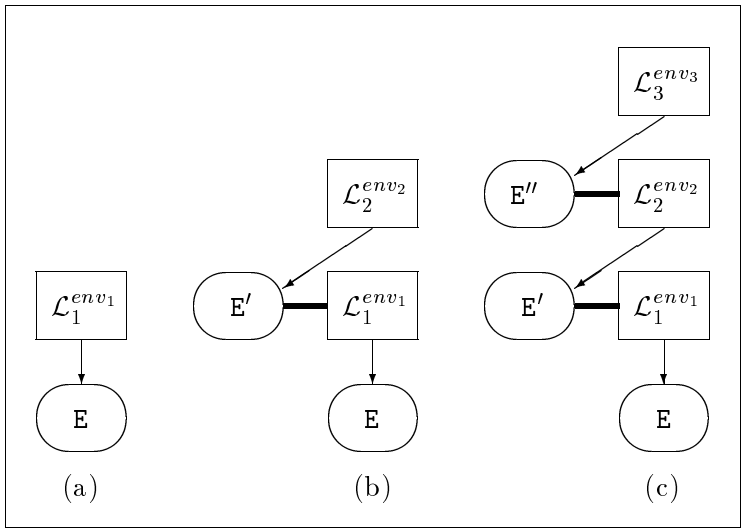

Figure 3.2: The approximate approach

program, and they might use different data representations. In Blond[21], this conversion is represented as the application of shifting operators $^{\wedge}$ and $^{\vee}$. 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.

3.1.3 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 $\mathcal{L}_{n+1}^{env_{n+1}}$ using partial evaluation. Although this method is interesting *per se* as the partial evaluation technique develops, it still involves various difficulties to go through. Right now, we construct a more static system where no dynamic compilation is performed. Later in Chapter 6, we will actually try to apply the partial evaluation techniques to obtain $\mathcal{L}'^{env_n}_n$ from $\mathtt{L}'^{env_n}_n$ and $\mathcal{L}_{n+1}^{env_{n+1}}$.

The approach we take here is as follows: 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

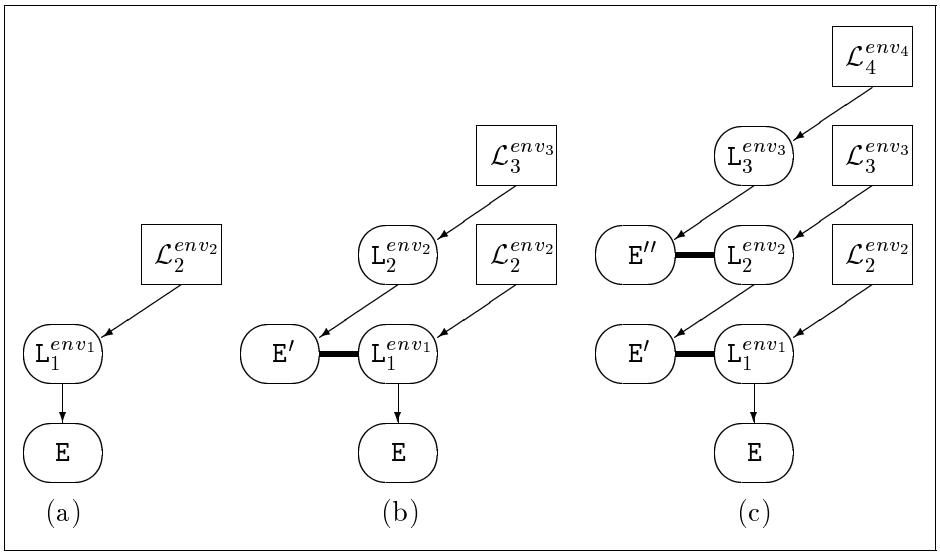

Figure 3.3: Our approach

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[1], which we will discuss in Section 3.2.

Figure 3.3 illustrates our approach before the partial evaluation is applied. Initially, a user program `E` is doubly interpreted by two interpreters, $\mathtt{L}_1^{env_1}$ and $\mathcal{L}_2^{env_2}$ (Figure 3.3(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 3.3(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 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.

3.1.4 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

[1]This partial evaluation is performed only conceptually when building the Black system. Once it is created, no dynamic compilation using partial evaluation will be done until Chapter 6.

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 3.3(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 3.4. 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 3.2.2.

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.

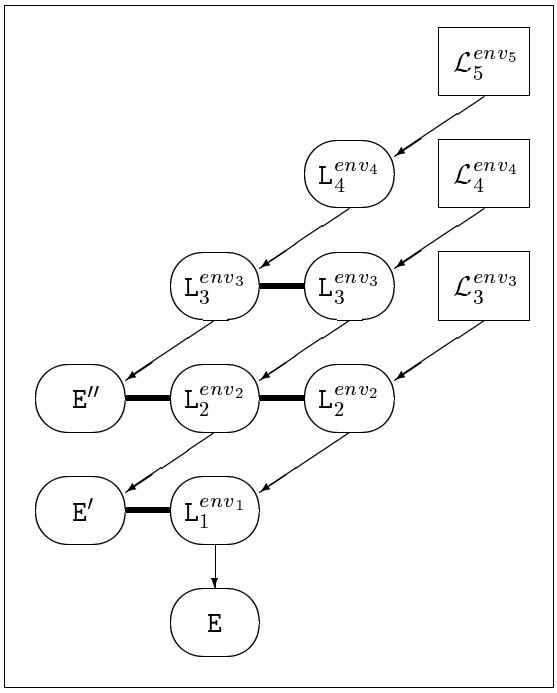

Figure 3.4: Using three interpreters for approximation

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.

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.

Another advantage of the two level approximation is that it contains sufficient information toward the exact implementation of the infinite tower and serves as a good starting point for it. The approximation becomes inaccurate when the environment env_{n+1} containing bindings for $\mathtt{L}_n^{env_n}$ and the environment which is implicitly used (in the underlying Scheme) to define $\mathcal{L}_n^{env_n}$ become inconsistent. By detecting the inconsistency and dynamically constructing directly executed interpreters using a partial evaluator as a dynamic compiler, we can faithfully implement the infinite tower without suffering from double interpretation.

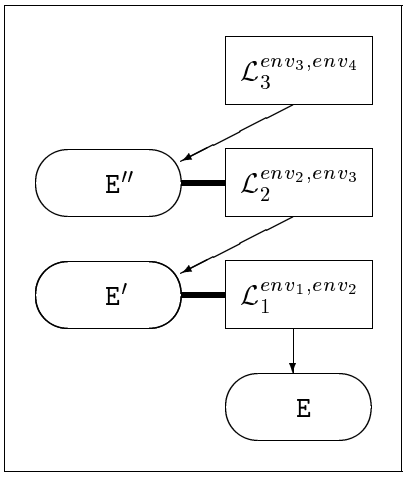

Figure 3.5: The tower of interpreters after the partial evaluation

3.2 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 $\mathcal{L}_{n+1}^{env_{n+1}}(\mathtt{L}_n^{env_n}(\mathtt{E}))$. In this section, we employ partial evaluation techniques[32] to remove this double interpretation. We assume here that readers are somewhat familiar with partial evaluation. Those who are not will find an introduction to partial evaluation in Chapter 5.

The scenario of this section is as follows. Knowing that $\mathcal{L}_{n+1}^{env_{n+1}}$ (other than env_{n+1}) does not change, 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 3.5 shows the structure of the tower after the partial evaluation. This is in contrast to Figure 3.3(c), in which user programs are always interpreted by two interpreters. Note that unlike Figure 3.2(c), $\mathcal{L}_n^{env_n,env_{n+1}}$ allows redefinitions of evaluator functions. Operationally, Figure 3.5 is exactly the same as Figure 3.3(c).

The partial evaluation we will perform is a manual one due to several technical reasons (see Section 3.2.3). 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 Chapter 4.

3.2.1 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[32]. 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* program) 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[2], 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 $\mathcal{L}_{n+1}^{env_{n+1}}$.

3.2.2 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}`))`[3]; 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

[2]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 $\mathtt{L}_n^{env_n}$ 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 4.3.

[3]We ignore the case in which `base-eval` is unbound in env_{n+1}.

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 $\mathrm{L}_{n+1}^{env_{n+1}}$, 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 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[4]. In Chapter 4, 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 could also 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[5].

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.

3.2.3 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.

[4]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.

[5]Suppose we want to make a trace of variable accesses.

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, 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 which we will discuss in Chapter 6).

Chapter 4

Implementation of Black

We shall now present the concrete implementation of Black in Scheme. This chapter together with the Appendix A gives the complete implementation of Black. In previous chapter, 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 employ is to lazily create a *metacontinuation* instead of $\mathcal{L}_n^{env_n, env_{n+1}}$ itself. The metacontinuation, proposed in Brown[27, 48] and later refined in Blond[21], 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.

4.1 Metacontinuation

A metacontinuation is an infinite list of pairs of an environment and a continuation[21].

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

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 `cons`ing 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 4.5 for the precise definition of `init-Mcont`.)

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

Using this definition, shifting one level up is 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.

4.2 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 4.1 (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 3.2.2 because of metacontinuations. (See Section 4.4 for more details.)

Note that Figure 4.1 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[1]. Metacontinuations are merely used to implement Black in Scheme, and are not visible from user programs, even if levelshifting occurs.

4.3 Evaluator Functions with Levelshifting

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

```
; 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))))
```

[1]Or more correctly, the *compiled* version of it, whose internal behavior cannot be modified by changing its metalevel interpreter.

```
; 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 4.1: Evaluator functions without levelshifting

Notice that `eval-EM` receives a metacontinuation `Mcont` explicitly, from which it extracts the metalevel environment, continuation, and the metacontinuation of the level above[2]. 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 3.2.2) 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))
               ((primitive? operator)
                (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 $\mathrm{L}_n^{env_n}$ (except for the insertion of `meta-apply`'s). The major difference related to levelshifting is in the `else` clause

[2]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.

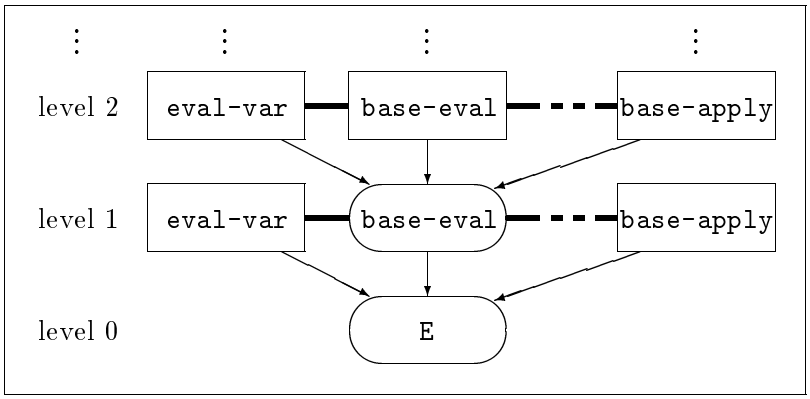

Figure 4.2: The tower structure when `base-eval` of level 1 is replaced

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 4.2, 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.

4.4 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 (primitive? operator)
                 ((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[3].

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`:

[3]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 Section3.2.2. This is justified because it can *never* be unbound: we use `meta-apply`'s exclusively for the application of evaluator functions which are registered in `init-env`.

```
(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.

4.5 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 2.3. 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.

4.6 Example Execution

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

```
> (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. 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 4.3), 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>
```

```
(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 4.3: New special forms: `break` and `inspect` in `break.blk`

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[4].

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

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.

Let us now modify the semantics of the language more drastically. We change the evaluation strategy from eager to lazy. The program is presented in Appendix B. It redefines `eval-application`, so that whenever a λ-expression is applied, its arguments are wrapped as thunks. Thunks are *forced* when they are applied to primitive procedures. Side-effects are used to memoize the value of thunks to prevent them from being evaluated more than once. `Init-cont` is also redefined to force returned values before they are printed.

To start afresh, we exit to level 2 and load lazy evaluation there. Suppose that the program is stored in `lazy.blk`:

```
1-5> (exit '(see you again))
2-0: (see you again)
2-1> (exec-at-metalevel (load "lazy.blk"))
New level loaded.
2-1: done
2-2>
```

Now, all the arguments to λ-closures are evaluated lazily. Thus, we can, for example, define lazy cons using λ-expressions:

```
2-2> (define (my-cons x y) (lambda (f) (f x y)))
2-2: my-cons
2-3> (define (my-car x) (x (lambda (x y) x)))
2-3: my-car
2-4> (define (my-cdr x) (x (lambda (x y) y)))
2-4: my-cdr
2-5> (define (integers-from n) (my-cons n (integers-from (+ n 1))))
2-5: integers-from
2-6> (define integers (integers-from 0))
2-6: integers
2-7> (my-car (my-cdr integers))
2-7: 1
2-8>
```

[4]Even if `eval-inspect` were installed at level 0, an error would occur for this expression, because `eval-inspect` in Figure 4.3 assumes that its argument is always a lambda closure.

4.7 Implementing the Delta Abstraction

The implementation framework presented in Chapter 3 is not limited to constructing Black. Here, we define a different language, which has the *delta* abstraction introduced by Blond[21] 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[5]:

```
(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[21].

4.7.1 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)))))))
```

[5] Assuming that continuations are 'pushy'[21].

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)
```

4.7.2 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 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 3.1.2.

Chapter 5

Partial Evaluation

In the previous chapters, we have constructed a reflective language Black. We saw that it satisfies the two extreme requirements: Black allows us to modify the language semantics (almost) in an *arbitrary* way at *any* level. Furthermore, it is realized in an efficient way, where directly executable evaluator functions are available not only for accesses but also for modification.

Although the Black interpreter executes user programs efficiently at first, user modification on the metalevel interpreter introduces double interpretation for modified parts. This considerably degrades the efficiency of Black, if the language semantics is modified in a substantial manner. Another problem is that at least the single interpretation is inherent, even though double interpretation is usually avoided. Since changes on the metalevel interpreter propagate to user programs through the sharing of a metalevel environment and the interpretation using the modified environment, user programs have to be interpreted if we want to make the changes effective.

To remove such interpretation overhead, we use partial evaluation[32]. It is well known that a partial evaluator can compile a program when an interpreter defining the language semantics is given[28]. Since the language semantics changes in reflective languages, compilers based on a specific language semantics are not permissible. Using partial evaluation, the compilation is performed according to the current language semantics.

In this chapter, we concentrate on designing a partial evaluator for Scheme, which is powerful enough to compile programs in Black[1]. Specifically, we will construct a partial evaluator which handles side-effecting operations correctly and preserves pointer equality. These facilities are important in compiling Black programs, since they play an essential role in the Black interpreter. Apart from side-effecting operations, our partial evaluator reduces more expressions than the previous one thanks to the proper treatment of pointer equality. After some simple extension, the resulting partial evaluator accepts almost full Scheme.

[1]For more details on the compilation framework of Black, see Chapter 6.

5.1 Introduction to Partial Evaluation

Given a program and parts of its arguments, a partial evaluator produces a *specialized* program with respect to those known arguments[32]. Since computation that depends only on the known arguments is performed at partial evaluation time, the resulting program usually runs faster at runtime than the original one. Partial evaluation is widely used not only as a general optimization tool[22] but also as various generators such as compiler generators[28] and parser generators[44].

To see how a partial evaluator specializes a program, consider the following traditional example program which computes the `n`th power of `m`:

```
(define (power m n)
  (if (= n 0)
      1
      (* m (power m (- n 1)))))
```

If we know beforehand that `n` has always a particular value, say three, we can symbolically execute the `power` program before receiving the actual value for `m` as follows:

```
(power m 3) = (if (= 3 0) 1 (* m (power m (- 3 1))))
            = (* m (power m 2))
            = (* m (if (= 2 0) 1 (* m (power m (- 2 1)))))
            = (* m (* m (power m 1)))
            = (* m (* m (if (= 1 0) 1 (* m (power m (- 1 1))))))
            = (* m (* m (* m (power m 0))))
            = (* m (* m (* m (if (= 0 0) 1 (* m (power m (- 0 1)))))))
            = (* m (* m (* m 1)))
```

Whenever arguments to a primitive procedure are all known, we *reduce* them. At the first line of the above trace, we reduced `(= 3 0)` to `#f` to take the *else* branch of `if`. When some of the arguments are unknown, on the other hand, we leave (or *residualize*) code that computes the result later when all the arguments are supplied. Since we do not known the value of `m` yet, the code to multiply `m` is residualized. If the symbolic execution reaches the application of user-defined functions, its body is *unfolded* to perform further specialization (if it does not lead to non-termination). By repeating these processes, we can obtain the final result (called a *residual* program) `(* m (* m (* m 1)))`. All the computation that depends on the known arguments are done at specialization time, and the resulting program efficiently computes the same answer as the original program when the same arguments are supplied.

When a partial evaluation is applied to an interpreter, we can *compile* a program. This is known as the first Futamura projection[28]. Consider the following simplified interpreter for Scheme written in direct style:

```
(define (base-eval exp env)
  (cond ((number? exp) exp)
        ((symbol? exp) (cdr (get exp env)))
        ((eq? (car exp) 'if)
         (if (base-eval (car exp) env)
             (base-eval (car (cdr exp)) env)
             (base-eval (car (cdr (cdr exp))) env)))
        ...
        (else
         (let ((lst (eval-list exp env)))
           (apply (car lst) (cdr lst))))))
```

If we specialize this interpreter with respect to a power program: `'(if (= n 0) 1 (* m (power m (- n 1))))`, we can obtain a compiled version of it as follows:

```
  (base-eval '(if (= n 0) 1 (* m (power m (- n 1)))) env)
= (if (base-eval '(= n 0) env)
      (base-eval '1 env)
      (base-eval '(* m (power m (- n 1))) env))
= (if (let ((lst (eval-list '(= n 0) env)))
        (apply (car lst) (cdr lst)))
      (base-eval '1 env)
      (base-eval '(* m (power m (- n 1))) env))
= (if (let ((lst (list (base-eval '= env)
                       (base-eval 'n env)
                       (base-eval '0 env))))
        (apply (car lst) (cdr lst)))
      (base-eval '1 env)
      (base-eval '(* m (power m (- n 1))) env))
= (if (let ((lst (list (get '= env) (get 'n env) 0)))
        (apply (car lst) (cdr lst)))
      (base-eval '1 env)
      (base-eval '(* m (power m (- n 1))) env))
= (if ((get '= env) (get 'n env) 0)
      (base-eval '1 env)
      (base-eval '(* m (power m (- n 1))) env))
= (if ((get '= env) (get 'n env) 0)
      1
      (base-eval '(* m (power m (- n 1))) env))
...
= (if ((get '= env) (get 'n env) 0)
      1
      ((get '* env) (get 'm env)
                    ((get 'power env) (get 'm env)
                                      ((get '- env) (get 'n env) 1))))
```

Observe that all the interpretation overhead inherited from `base-eval` is removed except for accesses to `env`, and the original expression, which was merely a data representing a program, is compiled into executable code. That is, a partial evaluator compiles a program given an interpreter defining its language semantics.

Partial evaluation of side-effecting constructs Although much work has been done on partial evaluation of functional languages, partial evaluation of side-effecting constructs remains difficult. In conventional partial evaluators for functional languages, only limited kinds of side-effects are allowed; other types are either merely refused or incorrectly processed. For example, Similix[13] handles only I/O operations and limited types of variable assignments. POPE[39] accepts richer kinds of variable assignments, but does not handle destructive updates of data structures correctly. This is a severe restriction since we often use side-effects even in mostly functional programs. In Black, for example, we use `set-cdr!` to change bindings in the environment.

In this chapter, we present an online partial evaluator for a call-by-value λ-calculus with destructive updates of data structures. It has the following novel characteristics:

- It correctly deals with full side-effects (including destructive updates of data structures).
- Pointer equality ('eq-ness') is preserved for all data structures.

These features are realized by a side-effect analysis and a systematic use of *preactions*[34, 35]. The preaction mechanism also enables us to reduce more expressions than the conventional let-expression approach[14, 11] used in Similix. Based on this framework, we create a partial evaluator for almost full Scheme[17].

Why side-effects are difficult to handle? The difficulties for partially evaluating programs with side-effects mainly come from mutable data structures (and mutable variables). Because partial evaluators perform non-standard eager evaluation, we need knowledge on the runtime value of data structures. In pure functional languages, the property of referential transparency assures that any data structure created at partial evaluation time has the same value throughout the specialization (and at runtime). With side-effects, this assumption does not hold any more, because assignment operations in residual code might modify the value of data structures.

The most conservative approach to handle mutable data structures is to treat all data structures as mutable and residualize them. This is unacceptable in practice, however, since the actually updated parts are usually small and the results become too conservative. Another approach would be to find mutable data structures during specialization. However, this does not work well either in the presence of first-class pointers. For example, consider partial evaluation of expressions that contain: `(set-car! x y)` where `x` is unknown. Without knowing the possible values for `x`, we have to treat all the data structures as unknown after this assignment, resulting in poor specialization.

Our approach is to identify mutable data structures (possible values for `x` in the above example) using a side-effect analysis. It extracts functional parts from programs with side-effects. Once mutable and immutable data structures are separated, partial evaluation is done in such a way that accesses to immutable ones are performed at specialization time, while accesses to mutable ones are residualized.

The separation is effective in practice, because most programs modify only restricted parts of data structures, and thus the separated mutable parts are usually small. Most parts are classified as immutable and we can achieve sufficient specialization. For example, consider

```
(lambda (x y z)
  (letrec ((inc (lambda (node visit)
                  (cond ((null? node) visit)
                        ((memq node visit) visit)         ; equality test
                        (else
                         (set-car! node (+ (car node) 1))        ; update
                         (inc (car (cdr node))
                              (inc (cdr (cdr node))
                                   (cons node visit))))))))))
    (let* ((make-node (lambda (value left right)
                        (cons value (cons left right))))
           (node1 (make-node x '() '()))
           (node2 (make-node y node1 '()))
           (node3 (make-node z node1 node2)))       ; node1 is shared !!!
      (inc node3 '())
      node3)))
```

Figure 5.1: An Example Program

an environment of interpreters for imperative languages, which is realized by an association list and destructive updates on it. The mutable parts are usually only the value slots of the environment and all other parts (including the overall structure of the environment) are immutable. When specializing this interpreter, we can unfold all the environment accesses, removing interpretation overhead completely.

Another difficulty for handling side-effects exists in residualization. We cannot eliminate, duplicate, or reverse code, since it might eliminate, duplicate, or reverse side-effects. These problems are avoided by the use of *preactions*[34, 35]. Without code elimination and duplication, our partial evaluator naturally preserves pointer equality of each value.

The preaction mechanism also enables us to treat data structures as both dynamic and static, i.e., we can use the value of data structures while residualizing them at the same time. This is a crucial property in reducing data structures whose identity has to be preserved. In conventional let-expression approach used in Similix, such data structures were made dynamic and put into let-expressions. However, this is an overly general approach for languages with pointer equality (such as Scheme), since *all* data structures have their identity and thus have to be residualized as dynamic expressions. The preaction mechanism enables us to residualize them without making them dynamic.

Example To see how our partial evaluator handles side-effecting operations, consider a Scheme program[2] shown in Figure 5.1. Given a DAG (directly acyclic graph) made by `make-node`, it increments the number associated to each node by one using a destructive update operation `set-car!`. To avoid multiple increment on shared nodes, visited nodes are

[2]We handle partial evaluation of closed terms only. When we want to specialize $f(x, y)$ with respect to $x = a$, we specialize $\lambda y.f(a, y)$ instead.

recorded in the argument `visit`, and checked via the predicate `eq?`[3]. To partially evaluate this program, we first perform the side-effect analysis. It will detect that the argument `node` of `inc` can be bound to cons cells constructed at the underlined `cons` statement in `make-node`. Because the *car* part of `node` is destructively updated in `inc`, the *car* parts of these underlined cells are mutable. The partial evaluator will residualize all accesses to these parts preserving the order of execution, and process other parts of the program. Because our partial evaluator reduces partially static data structures[37, 38], the *cdr* part of `node` is still accessible. After unfolding the iteration in `inc` (with a definition of `memq`), the result will look like this:

```
(lambda (x y z)
  (let* ((node1 (cons x (cons '() '())))
         (node2 (cons y (cons node1 '())))
         (node3 (cons z (cons node1 node2))))
    (set-car! node3 (+ (car node3) 1))
    (set-car! node2 (+ (car node2) 1))
    (set-car! node1 (+ (car node1) 1))
    node3))
```

All the references and updates to the mutable parts are residualized in the correct order, and other parts are completely unfolded. The names `node1`, `node2`, and `node3` are required to hold the identity (for pointer equality) of the three mutable cells. Notice here that even though these mutable cells declare their identity and are residualized in let-expressions, their static elements are reduced at specialization time. We can observe that `inc` is unfolded for the *cdr* part of `node3`, although `node3` itself is residualized in a let-expression. This distinguishes itself from the let-expression of Similix, where once an expression is residualized in a let-expression, it becomes completely dynamic. The use of preaction enables us to reduce such dynamic data structure that has static elements in it.

5.2 The Core Language

Towards a partial evaluator for Scheme, we first define a simple core language to concentrate on the treatment of side-effects. Its extension to Scheme is discussed in Section 5.6. The core language we use here is a call-by-value λ-calculus with mutable cons cells. The syntax is given as follows:

$$M = x \mid \eta : \lambda x.M \mid MN \mid \epsilon : consMN \mid carM \mid cdrM \mid setcarMN \mid setcdrMN \mid eqMM'NN'$$

An expression is either a variable, a λ-expression, an application, or one of six primitive expressions: *Cons* constructs a pair, whose elements are accessed by *car* and *cdr*. *Setcar* and *setcdr* destructively update an element of a pair. *Eq* is used to test the 'eq-ness' of values. It first evaluates its first two arguments to see if they are allocated in the same location. If they are, *eq* evaluates the third argument, discarding the fourth. Otherwise, it evaluates the fourth, discarding the third.

[3] `Eq?` is used in `memq`, whose definition is omitted here.

Each λ-expression and *cons* expression is attached with a unique label η and ϵ called a *closure point* and a *cons point*[19], respectively. These are used in the side-effect analysis in Section 5.3.

We assume that all bound variables are distinct. We write x^η when we explicitly show that the variable x is bound by a λ-expression labeled with η.

5.3 Side-effect Analysis

The goal of this section is to obtain two sets: $mutable_{car}$ and $mutable_{cdr}$, which contain cons points for mutable cells. In the partial evaluator presented in the next section, they are used to residualize accesses to mutable cells. If a cons point ϵ is in $mutable_{car}$, it means that the *car* part of the cons cells constructed by $\epsilon : cons$ might be modified by *setcar*. For the program in Figure 5.1, $mutable_{car}$ contains a cons point for the underlined `cons` expression and $mutable_{cdr}$ is empty. Since it is difficult to obtain the smallest $mutable_{car}$ and $mutable_{cdr}$, we obtain their safe approximation using an abstract-interpretation based approach similar to the Consel's[19].

The analysis consists of two stages. At the first stage, we collect all possible values (cons points and closure points) for each subexpression of a given input program. This is the central part of the analysis, which is achieved by the fixed-point iteration of abstract interpretation. It is a higher-order, interprocedural, context-insensitive, 0CFA[42] analysis.

Given the possible values for each subexpression, $mutable_{car}$ and $mutable_{cdr}$ are obtained fairly easily at the second stage. For each occurrence of $setcar\,M\,N$ in the program, we add the possible cons points for M to $mutable_{car}$. $Mutable_{cdr}$ is obtained similarly.

5.3.1 Notation

Abstract Value During abstract interpretation, each subexpression is assigned an abstract value A_v, which has a form $\langle \tau, \varphi \rangle$, where τ is a set of cons points and φ is a set of closure points. An abstract value represents the possible values for expressions. When an expression M has an abstract value $\langle \{\epsilon_1, \ldots, \epsilon_n\}, \{\eta_1, \ldots, \eta_m\} \rangle$, it means that M's value can be a cons cell constructed at one of ϵ_i's or a closure constructed at one of η_j's.

Abstract values are ordered by component-wise set-inclusion: $\langle \tau_1, \varphi_1 \rangle \sqsubseteq \langle \tau_2, \varphi_2 \rangle \Leftrightarrow (\tau_1 \subseteq \tau_2) \wedge (\varphi_1 \subseteq \varphi_2)$. The least upper bound of two abstract values is defined by component-wise set-union: $\langle \tau_1, \varphi_1 \rangle \sqcup \langle \tau_2, \varphi_2 \rangle = \langle \tau_1 \cup \tau_2, \varphi_1 \cup \varphi_2 \rangle$.

Abstract Environment The fixed-point iteration of abstract interpretation proceeds by updating an *abstract environment* ρ, a total function which maps variables to their abstract values. It has a functionality: $Variable \rightarrow A_v$, where *Variable* includes all variables appearing in the input program as well as three special variables: $car(\epsilon)$, $cdr(\epsilon)$, and $ran(\eta)$. $Car(\epsilon)$ and $cdr(\epsilon)$ are variables representing the *car* part and the *cdr* part of ϵ, respectively, while $ran(\eta)$ represents the range of η.

Two abstract environments ρ_1 and ρ_2 are ordered $\rho_1 \sqsubseteq \rho_2$, if $\rho_1(v) \sqsubseteq \rho_2(v)$ for all variables v. The least upper bound is similarly defined: $\rho_1 \sqcup \rho_2 = \lambda v.\rho_1(v) \sqcup \rho_2(v)$. The initial environment ρ_ϕ is defined by $\lambda v.\langle \{\}, \{\} \rangle$.

$$
\begin{array}{rcrl}
 & & \mathcal{AI} & : \quad Exp \to AbsEnv \to (A_v \times AbsEnv) \\
\rho & \in & AbsEnv & : \quad Variable \to A_v
\end{array}
$$

$$
\begin{array}{l}
\mathcal{AI}[\![x]\!]\rho = (\rho(x), \rho) \\
\mathcal{AI}[\![\eta : \lambda x.M]\!]\rho = \textbf{let } (\langle \tau, \varphi \rangle, \rho') = \mathcal{AI}[\![M]\!]\rho \\
\qquad\qquad\qquad\qquad \textbf{in } (\langle \{\}, \{\eta\} \rangle, \rho' \sqcup \rho_\phi[\langle \tau, \varphi \rangle / ran(\eta)]) \\
\mathcal{AI}[\![MN]\!]\rho = \textbf{let } (\langle \tau_1, \{\eta_1, \ldots, \eta_n\} \rangle, \rho_1) = \mathcal{AI}[\![M]\!]\rho \\
\qquad\qquad\qquad\quad (\langle \tau_2, \varphi_2 \rangle, \rho_2) = \mathcal{AI}[\![N]\!]\rho_1 \\
\qquad\qquad\qquad \textbf{in } (\bigsqcup_i \rho_2(ran(\eta_i)), \rho_2 \sqcup (\bigsqcup_{i,j} \rho_\phi[\langle \tau_2, \varphi_2 \rangle / x^{\eta_i}])) \\
\mathcal{AI}[\![\epsilon : consMN]\!]\rho = \textbf{let } (\langle \tau_1, \varphi_1 \rangle, \rho_1) = \mathcal{AI}[\![M]\!]\rho \\
\qquad\qquad\qquad\qquad\quad (\langle \tau_2, \varphi_2 \rangle, \rho_2) = \mathcal{AI}[\![N]\!]\rho_1 \\
\qquad\qquad\qquad\qquad \textbf{in } (\langle \{\epsilon_i\}, \{\} \rangle, \rho_2 \sqcup \rho_\phi[\langle \tau_1, \varphi_1 \rangle / car(\epsilon_i)] \sqcup \rho_\phi[\langle \tau_2, \varphi_2 \rangle / cdr(\epsilon_i)]) \\
\mathcal{AI}[\![carM]\!]\rho = \textbf{let } (\langle \{\epsilon_1, \ldots, \epsilon_n\}, \varphi \rangle, \rho') = \mathcal{AI}[\![M]\!]\rho \textbf{ in } (\sqcup_i \rho'(car(\epsilon_i)), \rho') \\
\mathcal{AI}[\![setcarMN]\!]\rho = \textbf{let } (\langle \{\epsilon_1, \ldots, \epsilon_n\}, \varphi_1 \rangle, \rho_1) = \mathcal{AI}[\![M]\!]\rho \\
\qquad\qquad\qquad\qquad\quad (\langle \tau_2, \varphi_2 \rangle, \rho_2) = \mathcal{AI}[\![N]\!]\rho_1 \\
\qquad\qquad\qquad\qquad \textbf{in } (\langle \tau_2, \varphi_2 \rangle, \rho_2 \sqcup (\bigsqcup_i \rho_\phi[\langle \tau_2, \varphi_2 \rangle / car(\epsilon_i)])) \\
\mathcal{AI}[\![eqMM'NN']\!]\rho = \textbf{let } (\langle \tau_1, \varphi_1 \rangle, \rho_1) = \mathcal{AI}[\![M]\!]\rho \\
\qquad\qquad\qquad\qquad\quad (\langle \tau_2, \varphi_2 \rangle, \rho_2) = \mathcal{AI}[\![M']\!]\rho_1 \\
\qquad\qquad\qquad\qquad\quad (\langle \tau_3, \varphi_3 \rangle, \rho_3) = \mathcal{AI}[\![N]\!]\rho_2 \\
\qquad\qquad\qquad\qquad\quad (\langle \tau_4, \varphi_4 \rangle, \rho_4) = \mathcal{AI}[\![N']\!]\rho_3 \\
\qquad\qquad\qquad\qquad \textbf{in } (\langle \tau_3 \cup \tau_4, \varphi_3 \cup \varphi_4 \rangle, \rho_4)
\end{array}
$$

Figure 5.2: Abstract Interpretation

5.3.2 The Abstract Interpretation

The rules for the abstract interpretation is shown in Figure 5.2. Given an expression M and the current abstract environment ρ, the abstract interpreter $\mathcal{AI}$ returns an abstract value for M and a possibly modified environment. For example, $\mathcal{AI}[\![car\ x]\!]\rho_\phi$ represents possible values for $car\ x$, where the value of x is not yet known. Thus, the result is $(\langle \{\}, \{\} \rangle, \rho_\phi)$. In the case of $\mathcal{AI}[\![car\ x]\!]\rho$ where $\rho(x) = \langle \{\epsilon\}, \varphi \rangle$ and $\rho(car(\epsilon)) = \langle \tau', \varphi' \rangle$, $\mathcal{AI}$ deduces that $car\ x$ can have the value $\langle \tau', \varphi' \rangle$, giving the result $(\langle \tau', \varphi' \rangle, \rho)$.

The rules in Figure 5.2 are fairly straightforward. We briefly sketch them here. The value of a variable is defined in ρ. The value of λ-expression is its attached closure point. Meanwhile, we analyze the body of the λ-expression under ρ to enlarge the environment and include the information on the range of the λ-expression (possible return values when the λ-expression is applied). This information is used in the next rule for applications. The value of an application is a union of its operators' range. The operand information is propagated to operators by adding it to the operators' bound variables. (This information is used in the next iteration when operators are analyzed.) Here, x^{η_i} is a parameter of the closure labeled with η_i. The rule for *cons* returns a cons point, leaving the abstract values of its *car* and *cdr* parts in the abstract environment. The rule for *car* collects the *car* part of its argument. The rule for *setcar* adds the possibility that the *car* part of its first argument may have the value of its second argument. Finally, the value of *eq* is a union of its then and else branches.

$$
\begin{array}{rcl}
Sval & = & \mathsf{var}(Var) \\
& | & \mathsf{closure}(ClosurePoint, Param, Body, Env, Name, PSval) \\
& | & \mathsf{apply}(Name, Name) \\
& | & \mathsf{pair}(ConsPoint, Name, Name) \\
& | & \mathsf{car}(Name) \\
& | & \mathsf{cdr}(Name) \\
& | & \mathsf{setcar}(Name, Name) \\
& | & \mathsf{setcdr}(Name, Name) \\
& | & \mathsf{eq}(Name, Name, PSval, PSval) \\
Preaction & = & Name : Sval \\
PSval & = & \langle\!\langle Preaction^* \rangle\!\rangle Name
\end{array}
$$

Figure 5.3: Symbolic Value and Preaction

The analysis is interprocedural, because the values of operands are propagated to the body of operators. It is context-insensitive, because the possible value of a bound variable is collected in one abstract value regardless of where it is originated from. It is 0CFA[42], because single closure point abstracts all the lambda closures created at that closure point.

5.3.3 Collecting Mutable Cells

Given an input program P, we repeatedly calculate $\mathcal{AI}[\![P]\!]\rho$ until ρ does not change, beginning with ρ_ϕ. Because abstract environments have only finite number of variations and operations on them are all monotonic, the environment will eventually reach a fixed-point ρ_{fix}, which captures all the possible values for variables. Using this, we obtain $mutable_{car}$ and $mutable_{cdr}$ as follows. Let $(\langle \tau_i, \varphi_i \rangle, \rho_{fix}) = \mathcal{AI}[\![M_i]\!]\rho_{fix}$ for the i'th occurrence of $setcar M_i N_i$ in P[4]. Since τ_i represents all the possible cons points for M_i, $mutable_{car}$ is a union of these cons points: $mutable_{car} = \bigcup_i \tau_i$. $Mutable_{cdr}$ is obtained similarly.

5.4 Partial Evaluation

We now present a partial evaluator for the core language. It is online, and interprets input programs recursively as interpreters do. When it encounters a primitive application, it takes one of two actions: reduce or residualize. If all the arguments to the primitive are known values, it reduces the application and returns a result. If some of the arguments are unknown, it returns an expression that performs the primitive application at runtime.

5.4.1 Symbolic Value and Preaction

To distinguish a value from code, online partial evaluators use symbolic values (Sval)[40]. The symbolic values we use are shown in Figure 5.3. There are nine Svals, each corresponding to the nine syntactic category of the core language. At the codification phase (Section 5.4.4),

[4]Since ρ_{fix} is a fixed-point, $\mathcal{AI}$ will always return ρ_{fix}.

they are translated into the corresponding piece of code. Among them, Svals for closures and pairs are *known* Svals: closures are unfolded when applied and pairs are referenced by *car* and *cdr*.

In the presence of side-effects, several issues that did not appear in pure functional languages become important, such as code elimination, code duplication, and the order of execution. To handle these issues, we introduce PSval, a symbolic value with a sequence of *preactions*[34, 35]. Preactions hold all the operations that has to occur when the symbolic value is created.

In the conventional online partial evaluators such as Fuse[49, 40], code duplication was avoided by using the graph structure of symbolic values. Although the graph structure is useful to separate the specialization process from the code duplication problem, it is complicated to translate the graph structure into concrete code preserving the sharing relationship between symbolic values. Rather than using the graph structure, we introduce indirection and use *names* to achieve the same effect. Readers can regard them as 'locations' of Svals. Then, a sequence of preactions acts like a store during partial evaluation. Names not only simplify the codification phase, but also make the structure of the whole partial evaluator much clearer, which enables us to prove its correctness.

5.4.2 Basic Strategies

Before presenting the partial evaluator, we demonstrate with examples how preactions are used to handle the above-mentioned issues, as well as how they handle side-effects and preserve eq-ness of values. It is instructive to know that a sequence of preactions will be coded as a sequence of *let* expressions.

Code Elimination Consider a program[5] $car(consMN)$, where N is a side-effectful expression. Reducing this program into M is incorrect because the side-effect that has to occur in N disappears. Instead, we save it as a preaction using a fresh name t as in $^{\langle\langle t:N\rangle\rangle}M$, which is coded into $let\ t = N\ in\ M$[6].

Code Duplication Code duplication occurs when a bound variable is used more than once. For example, assuming that N is side-effectful, reducing $(\lambda x.cons\, x\, x)N$ into $consNN$ is incorrect because the side-effect in N occurs twice. In such a case, we save N as a preaction with a fresh name t and refer to it by its name: $^{\langle\langle t:N\rangle\rangle}\mathsf{pair}(t,t)$, which is coded into $let\ t = N\ in\ cons\, t\, t$.

Order of Execution The order of side-effects has to be preserved. Consider a program $(\lambda x.\lambda y.cons\, y\, x)MN$, where both M and N are side-effectful. Reducing the program into $consNM$ reverses the order of M and N. Instead, we save M and N as preactions in the order of their evaluation: $^{\langle\langle t_1:M,t_2:N\rangle\rangle}\mathsf{pair}(t_2,t_1)$, which is coded into $let\ t_1 = M\ in\ let\ t_2 = N\ in\ cons\, t_2\, t_1$. Notice that a sequence of preactions holds a kind of execution history.

[5] In this subsection, we omit cons points for better readability.

[6] We denote $let\ t = N\ in\ M$ as a syntactic sugar for $(\lambda t.M)N$.

Handling of Side-effects Side-effectful operations are handled by residualizing all definitions, references, and updates of mutable cells in the correct order. For example, consider a program:

$$\begin{array}{l} \mathit{let}\ x = \mathit{cons} M N \\ \mathit{in}\ \mathit{let}\ y = \mathit{setcar}\, x\, M' \\ \quad \mathit{in}\ \mathit{cons}(\mathit{cdr}\, x)(\mathit{car}\, x). \end{array}$$

Since we know from the analysis in Section 5.3 that the *car* part of x is mutable, its definition ($\mathit{cons} M N$), reference ($\mathit{car}\, x$), and update ($\mathit{setcar}\, x\, M'$) are residualized as preactions. The partial evaluation of the above program becomes:

$$\langle\!\langle t_1{:}M, t_2{:}N, t_3{:}\mathsf{pair}(t_1,t_2), t_4{:}\mathsf{setcar}(t_3,M'), t_5{:}\mathsf{car}(t_3)\rangle\!\rangle \mathsf{pair}(t_2, t_5),$$

which is coded into:

$$\begin{array}{l} \mathit{let}\ t_1 = M \\ \mathit{in}\ \mathit{let}\ t_2 = N \\ \quad \mathit{in}\ \mathit{let}\ t_3 = \mathit{cons}\, t_1\, t_2 \\ \quad\quad \mathit{in}\ \mathit{let}\ t_4 = \mathit{setcar}\, t_3\, M' \\ \quad\quad\quad \mathit{in}\ \mathit{let}\ t_5 = \mathit{car}\, t_3 \\ \quad\quad\quad\quad \mathit{in}\ \mathit{cons}\, t_2\, t_5. \end{array}$$

Observe that $\mathit{cdr}\, x$ is reduced even though the value of x (i.e. $\mathsf{pair}(t_1, t_2)$) is put into preactions. Although $\mathsf{pair}(t_1, t_2)$ is given a name t_3 and residualized, it is not a dynamic value. This is in contrast to the conventional let-expression used in Similix, where once an expression is residualized in a let-expression, it is completely dynamic, and the information on its static parts are lost.

Preserving Eq-ness of Values In pure functional languages, it was enough to test value equality. In impure languages, we also have to handle pointer equality (eq-ness). It is intuitively understood that eq-ness is preserved in the absence of code elimination and duplication. We can also see this by regarding a sequence of preactions as a *store*. For example, consider a program $(\lambda x.M)(\mathit{cons} N N')$, which is specialized into $\langle\!\langle t{:}\mathsf{pair}(N,N')\rangle\!\rangle M[t/x]$. Here, the unique name t serves as the location for $\mathit{cons} N N'$. To test the eq-ness of cons cells, we can compare them by their names. The preservation of eq-ness is formally shown by proving that the partial evaluator does not change the store semantics of input programs, which is discussed in Section 5.5.

5.4.3 The Partial Evaluator

We now present the partial evaluator for the core language (Figure 5.4). Given an expression, an environment ρ, and a *pe-store* γ, the partial evaluator $\mathcal{PE}$ returns a PSval together with a (possibly) modified pe-store. A pe-store maps names introduced so far to their bound Svals.

The rules in Figure 5.4 are written in such a way that all Svals created are saved as preactions to prevent code elimination. The underlined names (such as $\underline{t}$) denote fresh names. Side-effects on mutable cells are treated by residualizing their definitions (in the *cons* rule)

$$
\begin{array}{rcrcl}
 & & \mathcal{PE} & : & Exp \to Env \to PEStore \to (PSval \times PEStore) \\
\rho & \in & Env & : & Variable \to Name \\
\gamma & \in & PEStore & : & Name \to Sval
\end{array}
$$

$$
\begin{array}{l}
\mathcal{PE}[\![x]\!]\rho\gamma = (^{\langle\langle\rangle\rangle}\rho(x), \gamma) \\
\mathcal{PE}[\![\eta : \lambda x.M]\!]\rho\gamma = \textbf{let } (^{\langle\langle P\rangle\rangle}t'', \gamma') = \mathcal{PE}[\![M]\!]\rho[\underline{t}'/x]\gamma[\mathsf{var}(\underline{t}')/\underline{t}'] \\
\qquad\qquad\qquad\qquad\qquad\qquad s = \mathsf{closure}(\eta, x, M, \rho, \underline{t}', {}^{\langle\langle P\rangle\rangle}\, t'') \\
\qquad\qquad\qquad\qquad\qquad \textbf{in } (^{\langle\langle \underline{t}:s\rangle\rangle}\underline{t}, \gamma[s/\underline{t}]) \\
\mathcal{PE}[\![MN]\!]\rho\gamma = \textbf{let } (^{\langle\langle P_1\rangle\rangle}t_1, \gamma_1) = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\qquad\qquad\quad (^{\langle\langle P_2\rangle\rangle}t_2, \gamma_2) = \mathcal{PE}[\![N]\!]\rho\gamma_1 \\
\qquad\qquad\qquad\qquad\quad \textbf{in case } \gamma_2(t_1) \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad \mathsf{closure}(\eta, x, M', \rho', _, _) \;:\; {}^{\langle\langle P_1,P_2\rangle\rangle}\mathcal{PE}[\![M']\!]\rho'[t_2/x]\gamma_2 \\
\qquad\qquad\qquad\qquad\qquad \textbf{otherwise} \;:\; \textbf{let } s = \mathsf{apply}(t_1, t_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } (^{\langle\langle P_1,P_2,\underline{t}:s\rangle\rangle}\underline{t}, \gamma_2[s/\underline{t}]) \\
\mathcal{PE}[\![\epsilon : consMN]\!]\rho\gamma = \textbf{let } (^{\langle\langle P_1\rangle\rangle}t_1, \gamma_1) = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\qquad\qquad\quad (^{\langle\langle P_2\rangle\rangle}t_2, \gamma_2) = \mathcal{PE}[\![N]\!]\rho\gamma_1 \\
\qquad\qquad\qquad\qquad\qquad\quad s = \mathsf{pair}(\epsilon, t_1, t_2) \\
\qquad\qquad\qquad\qquad\quad \textbf{in } (^{\langle\langle P_1,P_2,\underline{t}:s\rangle\rangle}\underline{t}, \gamma_2[s/\underline{t}]) \\
\mathcal{PE}[\![carM]\!]\rho\gamma = \textbf{let } (^{\langle\langle P\rangle\rangle}t, \gamma') = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\qquad\quad \textbf{in case } \gamma'(t) \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad \mathsf{pair}(\epsilon, t_1, t_2) \;:\; \textbf{if } \epsilon \in mutable_{car} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then let } s = \mathsf{car}(t) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } (^{\langle\langle P,\underline{t}':s\rangle\rangle}\underline{t}', \gamma'[s/\underline{t}']) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } (^{\langle\langle P\rangle\rangle}t_1, \gamma') \\
\qquad\qquad\qquad\qquad\qquad \textbf{otherwise} \;:\; \textbf{let } s = \mathsf{car}(t) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } (^{\langle\langle P,\underline{t}':s\rangle\rangle}\underline{t}', \gamma'[s/\underline{t}']) \\
\mathcal{PE}[\![setcarMN]\!]\rho\gamma = \textbf{let } (^{\langle\langle P_1\rangle\rangle}t_1, \gamma_1) = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\qquad\qquad\quad (^{\langle\langle P_2\rangle\rangle}t_2, \gamma_2) = \mathcal{PE}[\![N]\!]\rho\gamma_1 \\
\qquad\qquad\qquad\qquad\qquad\quad s = \mathsf{setcar}(t_1, t_2) \\
\qquad\qquad\qquad\qquad\quad \textbf{in } (^{\langle\langle P_1,P_2,\underline{t}:s\rangle\rangle}\underline{t}, \gamma_2[s/\underline{t}]) \\
\mathcal{PE}[\![eqMM'NN']\!]\rho\gamma = \textbf{let } (^{\langle\langle P_1\rangle\rangle}t_1, \gamma_1) = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\qquad\qquad\quad\;\; (^{\langle\langle P_2\rangle\rangle}t_2, \gamma_2) = \mathcal{PE}[\![M']\!]\rho\gamma_1 \\
\qquad\qquad\qquad\qquad\quad \textbf{in if } \gamma_2(t_1) \text{ and } \gamma_2(t_2) \text{ are known} \\
\qquad\qquad\qquad\qquad\qquad \textbf{then if } t_1 = t_2 \textbf{ then } {}^{\langle\langle P_1,P_2\rangle\rangle}\mathcal{PE}[\![N]\!]\rho\gamma_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } {}^{\langle\langle P_1,P_2\rangle\rangle}\mathcal{PE}[\![N']\!]\rho\gamma_2 \\
\qquad\qquad\qquad\qquad\qquad \textbf{else let } (^{\langle\langle P_3\rangle\rangle}t_3, \gamma_3) = \mathcal{PE}[\![N]\!]\rho\gamma_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\;\; (^{\langle\langle P_4\rangle\rangle}t_4, \gamma_4) = \mathcal{PE}[\![N']\!]\rho\gamma_3 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\;\; s = \mathsf{eq}(t_1, t_2, {}^{\langle\langle P_3\rangle\rangle}t_3, {}^{\langle\langle P_4\rangle\rangle}t_4) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } (^{\langle\langle P_1,P_2,\underline{t}:s\rangle\rangle}\underline{t}, \gamma_4[s/\underline{t}])
\end{array}
$$

Figure 5.4: Partial Evaluator

and updates (in the *setcar* rule). As for references (in the *car* rule), we lookup $mutable_{car}$ to see whether it is safe to access the cell at specialization time.

Let us see the rules in detail. A variable is translated into its bound name. A λ-expression is partially evaluated to a closure Sval. A fresh name $\underline{t}$ is given for the closure which is used to test its eq-ness. When constructing a closure Sval, we eagerly partially evaluate the body of the λ-expression. This is because the closure might be left residual in the residual program, in which case we want to create a specialized version of the closure. It is possible to defer the partial evaluation of body expressions to the codification phase. (We do so in our implementation.) However, this makes the correctness proof difficult, because we cannot separate the specialization phase from the codification phase.

To partially evaluate an application MN, we first partially evaluate M and N. If M evaluates to a known closure, it is unfolded. Here, we write ${}^{\langle\langle P \rangle\rangle}\mathcal{PE}[\![M]\!]\rho\gamma$ to mean $({}^{\langle\langle P,P' \rangle\rangle}t, \gamma')$ where $({}^{\langle\langle P' \rangle\rangle}t, \gamma') = \mathcal{PE}[\![M]\!]\rho\gamma$. Observe that preactions from M and N are propagated to the result without disturbing the order of execution. If M evaluates to unknown code, on the other hand, the application is residualized.

The partial evaluation of a *cons* expression proceeds similarly. After partially evaluating its arguments, a pair Sval with a fresh name $\underline{t}$ is constructed. Since *cons* is non-strict on its arguments during partial evaluation, we can always construct a pair. Although the pair Sval is given a name $\underline{t}$, it does not mean that the pair becomes an unknown value. Instead, it is residualized with the name $\underline{t}$ and used at the same time in the subsequent partial evaluation[7] (if it is not mutable). Observe that the pair Sval constructed here is also put into the pe-store for the later use[8].

For a *car* expression, its argument is first partially evaluated to see if it is a known pair. If it is *not*, residual code to take its *car* part is constructed. Otherwise, the *car* part is basically extracted and returned. However, because the *car* part might be mutable and prohibited from accessing, we lookup $mutable_{car}$. If it turns out to be mutable, we construct code rather than returning its *car* part.

The rule for *setcar* is simple. It residualizes a piece of code to perform *setcar* at runtime. Finally, partial evaluation of *eq* is done by first partially evaluating its first two arguments. When they are both known Svals, one of two branches is selected according to whether their locations are the same. When one (or both) of the arguments are unknown, both branches are specialized to construct *eq* code.

5.4.4 Codification

Since $\mathcal{PE}$ produces its output as a PSval, we have to convert it into the core language to complete the partial evaluation. Thanks to the use of names, the codification is straightforward as shown in Figure 5.5. We only note that the name clash will never occur, because all names are uniquely chosen at the specialization phase.

[7] That is, it is "both reduced and left residual"[49].

[8] Although all cons cells are residualized to declare their identity, they do not necessarily appear in the residual program. Post-processing phase will remove those preactions that will never be referenced during the runtime evaluation of the residual program.

$$\mathcal{PD}[\![^{\langle\langle\rangle\rangle}t]\!] = t$$
$$\mathcal{PD}[\![^{\langle\langle t_1 : s_1, \ldots\rangle\rangle}t]\!] = let\ t_1 = \mathcal{D}[\![s_1]\!]\ in\ \mathcal{PD}[\![^{\langle\langle \ldots\rangle\rangle}t]\!]$$

$$\mathcal{D}[\![\mathsf{closure}(\eta, _, _, _, t, {}^{\langle\langle P\rangle\rangle}t')]\!] = (\eta : \lambda t.\mathcal{PD}[\![^{\langle\langle P\rangle\rangle}t']\!])$$
$$\mathcal{D}[\![\mathsf{apply}(t_1, t_2)]\!] = (t_1\ t_2)$$
$$\mathcal{D}[\![\mathsf{pair}(\epsilon, t_1, t_2)]\!] = (\epsilon : cons\ t_1\ t_2)$$
$$\mathcal{D}[\![\mathsf{car}(t)]\!] = (car\ t)$$
$$\mathcal{D}[\![\mathsf{setcar}(t_1, t_2)]\!] = (setcar\ t_1\ t_2)$$
$$\mathcal{D}[\![\mathsf{eq}(t_1, t_2, {}^{\langle\langle P_3\rangle\rangle}t_3, {}^{\langle\langle P_4\rangle\rangle}t_4)]\!] = (eq\ t_1\ t_2\ \mathcal{PD}[\![^{\langle\langle P_3\rangle\rangle}t_3]\!]\ \mathcal{PD}[\![^{\langle\langle P_4\rangle\rangle}t_4]\!])$$

Figure 5.5: Codification

5.5 Correctness of the Partial Evaluator

In this section, we will briefly outline the proof that the partial evaluator presented in the previous section preserves the store semantics of the input program. The complete proof is in Appendix C.

Because our partial evaluator consists of two parts (specialization $\mathcal{PE}$ and codification $\mathcal{PD}$), the correctness is also divided into two parts: the correctness of $\mathcal{PD}$ and $\mathcal{PE}$. As for $\mathcal{PD}$, we first define the semantics of PSval $\mathcal{E}_{PS}$ in such a way that the semantics of PSval is given by the semantics of its codified program. Then, we prove that $\mathcal{PD}$ actually preserves the semantics: $\mathcal{E}_{PS}[\![^{\langle\langle P\rangle\rangle}t]\!]\sigma = \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P\rangle\rangle}t]\!]]\!]\sigma$, where $\mathcal{E}$ is a standard store semantics of the core language. The proof is straightforward and done by induction on the number of $\mathcal{PD}$ applications required to codify the PSval $^{\langle\langle P\rangle\rangle}t$. After that, we prove the main theorem stating that $\mathcal{PE}$ preserves the semantics: if γ respects σ, $\mathcal{E}_{PS}[\![\mathcal{PE}[\![M]\!]\rho\gamma]\!]\sigma = \mathcal{E}[\![M]\!]\rho\sigma$. The proof is again by induction on the number of $\mathcal{PE}$ applications required to partially evaluate M.

5.6 Extensions

So far, we have considered the partial evaluation of the core language. In this section, we describe extensions to cope with more realistic languages such as Scheme. Given a mechanism for handling side-effects correctly, it is not difficult to incorporate other features of functional languages. The extensions presented below are actually implemented in our Scheme partial evaluator, which can now deal with almost all features of Scheme.

Variable Assignments and I/O Assignments to variables can be handled by translating them to updates on data structures. Instead of storing a value v to a mutable variable x, we store a cell containing the value v to x. References and updates to x are translated to references and updates to the cell, respectively. Alternatively, assignments to variables can be handled by finding mutable variables and residualizing their definitions, references, and updates. This process parallels to the treatment of mutable cells and is realized almost the same as mutable cells. Since mutable variables can be found by scanning the input program

once, the analysis phase for finding mutable variables becomes much simpler. In our current implementation, the latter approach is taken.

Input/output operations are handled by simply residualizing them as preactions.

Recursion and Termination Detection Given that mutable cells are residualized correctly as unknown values, the presence of side-effects has no influence on the termination detection. We can introduce recursion and any conventional termination detection algorithm without problems. In our current implementation, we employ user annotation (or *filters*[18, 20]) to avoid non-termination.

λ-expressions with Variadic Parameters As pointed out by Thiemann[47], it is easy to handle variadic parameters when pairs are treated as partially static data structures[38]. By explicitly supplying cons points for the implicitly introduced cons cells for variadic parameters, we can support variadic parameters including destructive updates to those cons points. We can also support a higher-order primitive *apply* through extending the side-effect analysis to cope with function application caused by *apply*.

Constants and Other Primitives Constants and first-order primitives are handled straightforwardly. Higher-order primitives other than *apply* (e.g., *map* and *assq*) are provided as ordinary user-defined functions. We have not yet considered partial evaluation of *call/cc*.

Post-processing The post-processing phase removes those preactions that will not referenced during the runtime evaluation of the residual program. It also performs code beautification for the better readability. It includes: folding a sequence of `let` expressions into a `let*` expression, inlining expressions that are used exactly once[9], etc.

[9]Care must be taken if it is harmless to reverse the order of execution. In particular, the order of references to mutable cells (e.g., $car\,x$) and updates on them ($setcar\,xM$) cannot be reversed.

Chapter 6

Compilation Framework

Because the language semantics changes in reflective languages, it is impossible to build a specific compiler based on a particular languages semantics. To compile programs in reflective languages, we need more powerful tools that take account of the current language semantics. Partial evaluation[32] is one of such tools. It is well known that specializing an interpreter with respect to a program yields a compiled version of it[28]. By specializing a (possibly modified) metalevel interpreter, we can compile programs under the modified language semantics. Since we had constructed a partial evaluator which is powerful enough to handle side-effecting operations in the previous chapter, now is the time to use it as a compiler for reflective languages.

In this chapter, we discuss the overall compilation framework and integrate the partial evaluator into the Black interpreter. The actual examples of compilation using partial evaluation appear in the next chapter. According to a kind of programs we are to compile, there are several levels of compilation:

1. User programs do not contain reflective constructs.
2. User programs modify the language semantics at the beginning, and the following programs are executed under the modified semantics.
3. User programs dynamically modify the language semantics during execution, but the possible modification is enumerable.
4. User programs dynamically change the way to change the language semantics. In other words, user programs dynamically create the metalevel code according to the state of the computation[1].

Among them, we consider the first two cases. Despite the use of partial evaluation, it is still difficult to handle the third and fourth cases where dynamic modification of language semantics occurs.

[1]This is impossible in Black, since we did not include `primitive-EM` in the language. See Section 2.3.

6.1 Expression to be Partially Evaluated

Figure 6.1 shows the typical situation we will consider. In Figure 6.1(a), `base-eval` has been replaced with a user-defined function. To compile a user program `E` under this modified semantics, we first specialize the level 2 interpreter with respect to the replaced `base-eval` to obtain its compiled version `base-eval`$'$ (Figure6.1(b)). The result is a directly executed evaluator function which passes around the metacontinuation. Then, we specialize the level 1 interpreter (which includes the new `base-eval`$'$) with respect to a user program `E`. The resulting program `E`$'$ is an evaluator function compiled under the modified language semantics defined by the level 1 interpreter (Figure 6.1(c)).

The use of partial evaluation for compilation gives us two observations. The first one is that a compiled program is now independent of its metalevel interpreter. Since we have already executed most of the tasks performed by the metalevel interpreter at partial evaluation time, changes on the interpreter after the partial evaluation no longer affects the compiled program. That is, we have *duplicated* the metalevel interpreter when we performed partial evaluation. Another observation is that the resulting program is semantically identical to the original program interpreted by the metalevel interpreter.

Let us now identify the expression to be partially evaluated. To compile a function of the form:

```
(lambda (a b ...) <body>)
```

we partially evaluate the following expression:

```
(lambda (a b ...)
  (lambda (base-Mcont)
    (let ((env (car (head base-Mcont)))
          (cont (car (cdr (head base-Mcont))))
          (Mcont2 (tail base-Mcont)))
      ((base-eval <body>
                  (extend env '(a b ...) (list a b ...))
                  cont)
       Mcont)))) ; Mcont2 is ignored !!!
```

There are three free variables in this expression: `base-eval`, `extend`, and `Mcont`. At partial evaluation time, their current value (i.e., the value when the specialization is performed) is used, so that the compilation takes place under the current language semantics.

Let us now see the above expression in detail. When it received its arguments, it evaluates `<body>` under an extended environment. Because the compiled program is executed at runtime one level lower than the current level where the compilation is performed, the metacontinuation `base-Mcont` it receives is the one for the level below. Thus, the environment and continuation with which `<body>` is evaluated are stored in the `head` of `base-Mcont`.

Notice here that `Mcont`, instead of `Mcont2`, is used for the specialization of `<body>`. If compilation were not performed, `<body>` would have been evaluated under `Mcont2`, the metacontinuation at the function's application time. However, because the value of `Mcont2` is

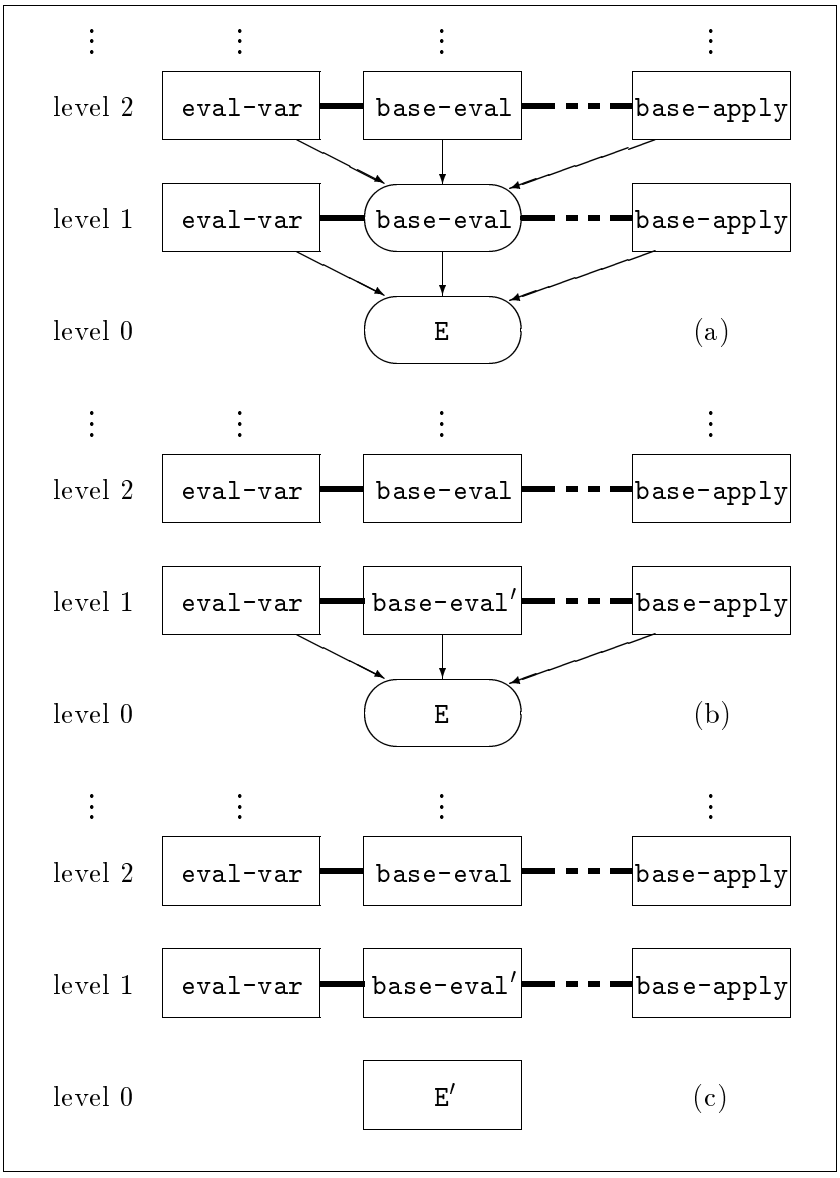

Figure 6.1: Overall scenario of the compilation

unknown at partial evaluation time, we use instead `Mcont`, the metacontinuation at the compilation time. That is, we *duplicate* the current metacontinuation and use it for compilation. This means that after the compilation, compiled programs are no more sensitive to the redefinition of their metalevel interpreter.

While the metacontinuation is duplicated and used during partial evaluation, `env` and `cont` are unknown values extracted from unknown `base-Mcont`. This means that the resulting compiled code respects the current value of variables and functions in the same level. Thus, if `eval-var` is replaced in Figure 6.1(b), the compiled `base-eval′` will correctly call the new `eval-var` afterwards.

6.2 Integrating Partial Evaluator into Black

To realize the compilation framework presented in the previous section, we now integrate a partial evaluator into Black. We will discuss several issues that have to be considered.

6.2.1 Source Code of Evaluator Functions

Partial evaluation is a source-to-source transformation technique. Usually, we assume that a partial evaluator is given an entire program in a file. When we compile a program in Black, however, the interpreter to be specialized is already running. Because the underlying Scheme does not usually provide us to extract source code from executable λ-closures, we have to supply it separately. This can be easily done, for example, by maintaining a table which holds the source code for all λ-closures appearing in Black.

The source code stored in the table is the code that is actually executed, not the user observable one. In particular, it contains manipulation of metacontinuations. Among them, the code for `eval-lambda` is somewhat special. In Figure 4.1, `eval-lambda` was defined as follows:

```
(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))))
```

When this function is called, it returns a list representing a lambda closure. As we saw in Section 3.2.3, this causes a trouble when we compile programs containing a lambda closure. Rather than compiling it into an actual lambda closure, our partial evaluator will produce a list representing a lambda closure. To avoid this, we artificially replace the code of `eval-lambda` for compilation with the following[2]:

[2]We replace only the source code that is used for partial evaluation. We use the original `eval-lambda` for normal execution.

```
(define (eval-lambda exp env cont)
  (lambda (Mcont)
    (let ((lambda-body (cdr (cdr exp)))
          (lambda-params (car (cdr exp))))
      ((meta-apply cont
                   (lambda args (lambda (base-Mcont)
                     (let ((meta-cont (car (cdr (head base-Mcont))))
                           (meta-env (car (head base-Mcont)))
                           (Mcont2 (tail base-Mcont)))
                       ((eval-begin lambda-body
                                    (extend env lambda-params args)
                                    meta-cont)
                        Mcont))))) ; Mcont2 is ignored !!!
       Mcont))))
```

The new code for `eval-lambda` returns a lambda-closure instead of a list. By duplicating and propagating the current metacontinuation `Mcont` (ignoring `Mcont2`), we can compile lambda-closures into lambda-closures as expected.

Notice that this replacement is rather ad-hoc. Because we know that the list representing a lambda-closure is always used as a lambda-closure in `base-apply`, we could replace the code for `eval-lambda`. If we redefine `base-apply`, however, this replacement is no longer valid. We have to introduce another mechanism to convert a data structure representing a lambda-closure into an actual lambda-closure, if we want it to be compiled into an actual lambda-closure.

6.2.2 Sharing Environments between Partial Evaluator and Black

The fact that the interpreter to be specialized is already running in Black causes another problem. Because some data structures whose identity have to be preserved (e.g., `lambda-tag`) are created before specialization. Partial evaluation has to be done in such a way that their identity is preserved during and after the specialization.

For this purpose, we *share* the environment between the partial evaluator and Black. We use the environment of Black interpreter as an environment of the partial evaluator, too. Because the Black interpreter and the partial evaluator store data in the environment differently, i.e., the interpreter stores immediate values whereas the partial evaluator stores Svals, we augment the partial evaluator to cope with immediate values. During specialization, immediate values are treated as ordinary known values. When they are residualized, we leave code that takes them from the environment, so that their identity is preserved. For example, assume that `lambda-tag` is bound to a cons cell declared as `(cons 'lambda 'tag)` in the environment. During specialization, it is treated as a known pair whose identity is determined by the actual address of the pair. When it residualizes, we use `(cdr (get 'lambda-tag env))` instead of `(cons 'lambda 'tag)`. The latter does not preserve the identity of `lambda-tag`, as it creates another cons cell.

Sharing environments also enables us to compile programs under the current language semantics without worrying about the way the metalevel interpreter is redefined. Users are allowed to use an arbitrary sequence of `set!` expressions to change the metalevel interpreter.

Although our partial evaluator does not statically reduce side-effecting operations, it can compile programs under the current language semantics by using the current environment of the Black interpreter as the environment of partial evaluation.

6.3 Binding-Time Improvement

Before we perform compilation using partial evaluation, we present a binding-time improvement required to specialize Black interpreter well. Since the interpreter to be specialized is written in metacontinuation passing style, propagation of metacontinuation is essential. Consider `eval-if`:

```
(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)))))
```

By η-expanding metacontinuation, we can write the continuation to the first call to `base-eval` as follows:

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

If `pred` evaluates to a known value, the specialization of this expression proceeds without any problems. When it evaluates to an unknown value, however, the partial evaluation of `then` and `else` branches does not take place, because `Mcont` is not applied to them. To propagate `Mcont` into `then` and `else` branches, we rewrite the above continuation as follows:

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

Chapter 7

Examples of Compilation

Based on the compilation framework presented in the previous chapter, we demonstrate several examples of compilation in this chapter to see how the framework works in practice. All the examples in this chapter include multi-level compilation except for the first one. They can be also regarded as a non-trivial application of partial evaluation.

7.1 Effect of Primitives

The first example we demonstrate is a tiny program which adds one to its argument: `(lambda (x) (+ x 1))`. We will see how an application of a primitive procedure is compiled. To compile this expression, we partially evaluate the following expression:

```
(lambda (x)
  (lambda (base-Mcont)
    (let ((env (car (head base-Mcont)))
          (cont (car (cdr (head base-Mcont))))
          (Mcont2 (tail base-Mcont)))
      ((base-eval '(+ x 1)
                  (extend env '(x) (list x))
                  cont)
       Mcont))))
```

Unfortunately, the result becomes 245 lines of code! The core part of it is shown in Figure 7.1. Why did the result become so large? — Because `env` is completely unknown. By carefully examining the result, we observe that it first looks up `env` to find a binding of `+`. Then, it dispatches according to the value bound to `+`. When it is a primitive, the expected result `(cont (operator-+ x 1))` is residualized. However, the result has another cases when `+` is an evaluator function, a user-defined lambda closure, a non-procedure (in which case an error will be reported), and even the case when `+` is unbound!

If we want the compiled code to be sensitive to the redefinition of primitive functions, the result is what we expected. We can no more specialize the result further under unknown `env`. The question is whether we want it or not. Here, we choose to specialize the result further by supplying information on primitives to `env`. Rather than using completely unknown `env`, we partially evaluate the following expression:

```
(lambda (x)
  (lambda (base-Mcont)
    (let* ((env (car (head base-Mcont)))
           (cont (car (cdr (head base-Mcont))))
           (Mcont2 (tail base-Mcont))
           ...
           (get-fun (lambda args
                      (apply (cdr (get 'get (car (head Mcont2)))) args)))
           (pair (get-fun '+ env)))
      (if (pair? pair)
          (let* ((operator-+ (cdr pair))
                 ...
                 (g-272 (if (procedure? operator-+)
                            (if (primitive? operator-+)
                                (cont (operator-+ x 1)) ; Addition !!!
                                (...)) ; When + is an evaluator function.
                            (if (if (pair? operator-+)
                                    (eq? (car operator-+) lambda-tag)
                                    #f)
                                (...)     ; When + is a lambda-closure.
                                (...))))) ; When + is not a procedure.
            (g-272 Mcont2))
          (...))))) ; When + is unbound.
```

Figure 7.1: The result of compiling `(lambda (x) (+ x 1))`

```
(lambda (x)
  (lambda (base-Mcont)
    (let ((env (car (head base-Mcont)))
          (cont (car (cdr (head base-Mcont))))
          (Mcont2 (tail base-Mcont)))
      ((base-eval '(+ x 1)
                  (extend (cons primitive-frame env) ; add primitives !!!
                          '(x) (list x))
                  cont)
       Mcont))))
```

`Primitive-frame` holds bindings for all the primitive procedures. By supplying them explicitly, we make them known at partial evaluation time. Now, the result becomes as expected:

```
(lambda (x)
  (lambda (base-Mcont)
    (let* ((cont (car (cdr (head base-Mcont))))
           (Mcont2 (tail base-Mcont))
      ((cont (+ x 1)) Mcont2)))))
```

Note that in Section 3.2.2, we have made a similar decision when we construct default evaluator functions of Black. Since primitive procedures are usually not redefined, we did not insert `meta-apply`'s for them. In the same way, by duplicating the value for primitives, we perform more specialization while making the compiled code insensitive to their redefinition.

7.2 Residualizing Calls to `base-apply`

As a second example, we consider the tracing `base-eval` shown in Section 2.4. Suppose that we saved the original `base-eval` in `original-eval` at level 1:

```
0-1> (exit 0)
New level loaded.
1-0: 0
1-1> (define original-eval base-eval)
1-1: original-eval
1-2>
```

Then, we execute the following expression:

```
1-2> (set! base-eval
           (compile (lambda (base-exp base-env base-cont)
                      (write base-exp)
                      (original-eval base-exp base-env base-cont))))
1-2: base-eval
1-3>
```

Here, `compile` transforms its argument into an appropriate expression to be specialized and performs partial evaluation. Assume that primitives are visible as in the previous section. Unfortunately, the result becomes quite similar to the previous example with more than 300 lines of code. Although the partial evaluator can transform `(write base-exp)` to itself completely removing the interpretation process, `original-eval` is unknown in `env`, resulting in the residualization of almost entire code of `base-apply`. We do not want here to make `original-eval` known in `env` as we did for primitive procedures in the previous section, since it is an evaluator function comprising the interpreter and is likely to be redefined later. If we made it known in `env`, the compiled code would become insensitive to the redefinition of `original-eval`.

The lengthy result is not necessarily harmful. It is faster than the original uncompiled code and does not prevent further compilation using it. We will actually use this long result in the following sections. However, when we look and examine it, shorter results are preferable.

To make the result somewhat shorter, we use the following heuristics: whenever an operator to be applied in `base-apply` is unknown, we do not unfold the call to `base-apply` but residualize it. Then, the result becomes as shown in Figure 7.2. Although a check whether `original-eval` is bound or not still remains, we can observe that the overall structure of the result is a `write` expression followed by an application of `original-eval`. Note that we have already used this technique before. In Section 3.2.2, we have abstracted similar code to form `meta-apply`'s. Residualizing calls to `base-apply` corresponds to using `meta-apply`'s.

Let us now go back to the baselevel and compile a program under the tracing `base-eval`:

```
1-3> (old-cont 0)
0-1: 0
0-2> (define f (compile (lambda (x) (+ x 1))))
0-2: f
0-3>
```

```
(lambda (base-exp base-env base-cont)
  (lambda (base-Mcont)
    (let* ((env (car (head base-Mcont)))
           (cont (car (cdr (head base-Mcont))))
           (Mcont2 (tail base-Mcont))
           (lst (list (cons 'base-exp base-exp)
                      (cons 'base-env base-env)
                      (cons 'base-cont base-cont)))
           (extended-env (cons lst env))
           (get-fun (lambda args
                      (apply (cdr (get 'get (car (head Mcont2))))
                             args)))
           (base-apply-fun (lambda args
                             (apply (cdr (get 'base-apply
                                              (car (head Mcont2))))
                             args))))
      (write base-exp) ; trace !!!
      (let* ((cont-10-443
                (lambda (operator)
                    (lambda (Mcont3)
                      (let* ((operand (list base-exp base-env base-cont))
                             (g-1069 (base-apply-fun operator
                                                     operand
                                                     extended-env
                                                     cont)))
                        (g-1069 Mcont3)))))))
             (pair (get-fun 'original-eval env)))
        (if (pair? pair)
            (let* ((original-eval-fun (cdr pair))
                   (g-793 (base-apply-fun original-eval-fun
                                          (list base-exp base-env base-cont)
                                          extended-env
                                          cont)))
              (g-793 Mcont2))
            (let* ((my-error-fun (lambda args
                                   (apply (cdr (get 'my-error
                                                    (car (head Mcont2))))
                                          args)))
                   (x-800 (list 'eval-var: 'unbound 'variable: 'original-eval))
                   (g-804 (my-error-fun x-800 extended-env cont-10-443)))
              (g-804 Mcont2)))))))
```

Figure 7.2: The result of compiling the tracing `base-eval`

The result becomes as follows:

```
(lambda (x)
  (lambda (base-Mcont)
    (let* ((cont (car (cdr (head base-Mcont))))
           (Mcont2 (tail base-Mcont)))
      (write (list '+ 'x 1))
      (write '+)
      (write 'x)
      (write 1)
      (let* ((x-1798 (list (car (head Mcont2))          ;
                           (car (cdr (head Mcont2)))))  ; levelshift overhead
             (Mcont3 (cons x-1798 (tail Mcont2))))      ;
        ((cont (+ x 1)) Mcont3)))))
```

All the interpretation overhead for printing traces is completely removed and a sequence of `write` expressions is residualized. The user program is correctly compiled under the modified semantics.

Although write expressions and `(+ x 1)` are fully compiled, the overhead of shifting levels remained in the result. It extracts the `head` part of `Mcont2` which is immediately stored back to `Mcont2`. This is because the compiled `base-eval` internally interprets user programs one level higher than the original `base-eval`. If the compilation were not performed, `write` expressions would have been interpreted by the level 2 interpreter. However, since all these processes performed at level 2 are done at specialization time, only shifting up and shifting down operations remain residualized.

7.3 Effect of Error-Handling

In the examples shown in the preceding sections, results of compilation tend to become large. The main reason comes from error-handling routines. Since the metalevel interpreter to be specialized contains these routines, it is natural that the residual program inherits them. However, it would not be preferable if the large residual programs result in even larger residual programs when compilation under the former are performed. Fortunately, we will see that this is not the case.

To see the effect of error-handling routines, we replace an evaluator function of the metalevel interpreter with a compiled function which has the same semantics as the original one. By comparing the results of compilation under the former and the latter, we can see how error-handling routines residualize in the compiled code. We replace, for example, `eval-list` here:

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

```
1-1> (set! eval-list
           (compile (lambda (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))))))))))
1-1: eval-list
1-2>
```

The new compiled `eval-list` is semantically identical to the original one, but contains error-handling routines inherited from the level 2 interpreter. Then, we compile a program under this modified `eval-list`:

```
1-2> (old-cont 0)
0-1: 0
0-2> (define f (compile (lambda (x) (+ x 1))))
0-2: f
0-3>
```

The result becomes as follows:

```
(lambda (x)
  (lambda (base-Mcont)
    (let* ((cont (car (cdr (head base-Mcont))))
           (Mcont2 (tail base-Mcont))
           (x-3683 (list (car (head Mcont2))          ;
                         (car (cdr (head Mcont2)))))  ; levelshift overhead
           (Mcont3 (cons x-3683 (tail Mcont2))))      ;
      ((cont (+ x 1)) Mcont3))))
```

In Section 7.1, the same program was compiled under the standard semantics, giving the following result:

```
(lambda (x)
  (lambda (base-Mcont)
    (let* ((cont (car (cdr (head base-Mcont))))
           (Mcont2 (tail base-Mcont))
      ((cont (+ x 1)) Mcont2)))))
```

By comparing these results, we observe that they are identical except for the levelshift overhead. All error-handling routines completely disappear. Although the compiled `eval-list` contains many error-checks in it, they check the errors of the metalevel and are all reduced at partial evaluation time. Only the error-checks of baselevel programs residualize, which, in this example, do not exist thanks to the unfolding of primitive procedures.

This example does not imply that all the error-checks are always reduced at specialization time. Rather, the error-checks of baselevel programs can remain residualized. For example, consider the following expression where `f` is unknown at compilation time:

```
0-3> (define g (compile (lambda (x) f)))
0-3: g
0-4>
```

Then, `g` contains an error-check to see if `f` is bound at baselevel or not:

```
(lambda (x)
  (lambda (base-Mcont)
    (let* ((env (car (head base-Mcont)))
           (cont (car (cdr (head base-Mcont))))
           (Mcont2 (tail base-Mcont))
           (lst (list (cons 'x x)))       ; additional overhead caused
           (extended-env (cons lst env))  ;                          by error-check
           (get-fun (lambda args
                      (apply (cdr (get 'get (car (head Mcont2))))
                             args)))
           (pair (get-fun 'f env)))
      (if (pair? pair)
          ((cont (cdr pair)) Mcont2)
          (let* ((my-error-fun (lambda args
                                 (apply (cdr (get 'my-error (car (head Mcont2))))
                                        args)))
                 (x-359 (list 'eval-var: 'unbound 'variable: 'f))
                 (g-363 (my-error-fun x-359 extended-env cont)))
            (g-363 Mcont2))))))
```

The result contains not only the baselevel error-check, but also an additional overhead caused by the error-check. Since `my-error` in Black is designed to store the environment and continuation when an error occurs, the result has to maintain all the intermediate environments and continuations. This causes the result to be larger and less efficient. It also makes the time for further partial evaluation using this result much longer. To avoid this, we can arbitrarily replace the code for `my-error` to ignore the environment and continuation. This would considerably reduce the code size of the result, although it introduces additional inconsistency between the actual code and the code for partial evaluation, as in `eval-lambda`.

7.4 Extending the Language

As a next example, we extend the baselevel language to cope with a `case` special form and see how the partial evaluator works for programs under this semantics. The program is shown in Figure 7.3.

```
0-1> (exec-at-metalevel (load "case.blk"))
New level loaded.
0-1: done
0-2> (define (fib n) (case n ((0 1) n)
                             (else (+ (fib (- n 1)) (fib (- n 2))))))
0-2: fib
0-3>
```

```
(define (member? key lst)
  (cond ((null? lst) #f)
        ((eq? key (car lst)) #t)
        (else (member? key (cdr lst)))))

(define (eval-case-clause key clauses env cont)
  (cond ((null? clauses) (cont '*unspecified*))
        ((eq? (car (car clauses)) 'else)
         (eval-begin (cdr (car clauses)) env cont))
        ((member? key (car (car clauses)))
         (eval-begin (cdr (car clauses)) env cont))
        (else
         (eval-case-clause key (cdr clauses) env cont))))

(define (eval-case key-exp clauses env cont)
  (base-eval key-exp env (lambda (key)
    (eval-case-clause key clauses env cont))))

(define original-eval-application eval-application)

(set! eval-application
      (lambda (exp env cont)
        (if (eq? (car exp) 'case)
            (eval-case (car (cdr exp)) (cdr (cdr exp)) env cont)
            (original-eval-application exp env cont))))
```

Figure 7.3: Black program to introduce `case` special form (`case.blk`)

We define the (exponential) fibonacci function `fib` using `case` and measure its performance. We examine three cases:

1. `fib` is interpreted under the modified interpreter (without compilation)
2. `fib` is interpreted under the compiled interpreter
3. `fib` is compiled under the compiled interpreter

The following transcript shows how we measure these cases:

```
0-3> (fib 17)                                          ; case 1
0-3: 1597
0-4> (exit 0)
1-0: 0
1-1> (set! member? (compile member?))                  ; compile
1-1: member?
1-2> (set! eval-case-clause (compile eval-case-clause))
1-2: eval-case-clause
1-3> (set! eval-case (compile eval-case))
1-3: eval-case
1-4> (set! eval-application (compile eval-application))
1-4: eval-application
1-5> (old-cont 4)
0-4: 4
0-5> (fib 17)                                          ; case 2
0-5: 1597
0-6> (set! fib (compile fib))                          ; compile
0-6: fib
0-7> (fib 17)                                          ; case 3
0-7: 1597
0-8>
```

The experiment is done using Chez Scheme on a SUN SS20 workstation (HyperSPARC 150MHz) with 128 Mbyte of memory. The result is summerized in Table 7.1. For comparison, we show figures for the fibonacci function `fib2` which does not use `case` and is executed under the standard semantics:

```
0-1> (define (fib2 n)
       (if (< n 2)
           n
           (+ (fib2 (- n 1)) (fib2 (- n 2)))))
0-1: fib2
0-2> (fib2 17)                          ; case 2
0-2: 1597
0-3> (set! fib2 (compile fib2))         ; compile
0-3: fib2
0-4> (fib2 17)                          ; case 3
0-4: 1597
0-5>
```

	execution time (sec.)	compile time (sec.)
`(fib 17)` case 1	120	-
2	11	17
3	2	31
`(fib2 17)` case 2	11	-
3	2	4

Table 7.1: Execution and compilation time for `(fib 17)`

Compared to the case 2 of `(fib2 17)`, `(fib 17)` under modified interpreter without compilation (case 1) is considerably slow due to the double interpretation. However, once the interpreter is compiled, it runs as efficient as `fib2`. We can also compile `fib` to obtain the speedup of factor about 6. The resulting compiled code is almost identical to the one obtained by compiling `fib2` under the standard semantics. Only difference is that the former tests `n=0` and `n=1` whereas the latter `n<2`. All the syntactic dispatches (including `case`) are completely compiled and programs are successfully compiled even under the modified semantics.

The compile time to obtain the compiled version of `fib` is a bit long, because the compiled metalevel interpreter contains error-handling routines and is large. These error-checks are all concerned with the metalevel errors and hence are reduced at partial evaluation time, making the compile time longer.

7.5 Lazy Evaluation

We are now ready to compile a bigger program. We will change the baselevel language to take the lazy evaluation strategy, compile a program under this semantics, and see how the compilation improves performance. The program to change the evaluation strategy is shown in Appendix B. We again examine three cases as we did in Section 7.4:

```
0-1> (exec-at-metalevel (load "lazy.blk"))
New level loaded.
0-1: done
0-2> (define (fib n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))
0-2: fib
0-3> (fib 17)                                          ; case 1
0-3: 1597
0-4> (exit 0)
1-0: 0
1-1> (set! thunk? (compile thunk?))                    ; compile
1-1: thunk?
1-2> (set! make-thunk (compile make-thunk))
1-2: make-thunk
1-3> (set! make-thunk-list (compile make-thunk-list))
1-3: make-thunk-list
1-4> (set! force (compile force))
```

	execution time (sec.)	compile time (sec.)
`(fib 17)` case 1	480	-
2	40	29
3	14	1387

Table 7.2: Execution and compilation time for `(fib 17)` under lazy evaluation

```
1-4: force
1-5> (set! force-list (compile force-list))
1-5: force-list
1-6> (set! eval-application (compile eval-application))
1-6: eval-application
1-7> (set! eval-if (compile eval-if))
1-7: eval-if
1-8> (set! init-cont (compile init-cont))
1-8: init-cont
1-9> (old-cont 0)
0-4: 0
0-5> (fib 17)                                          ; case 2
0-5: 1597
0-6> (set! fib (compile fib))                          ; compile
0-6: fib
0-7> (fib 17)                                          ; case 3
0-7: 1597
0-8>
```

The result is summerized in Table 7.2. As was in the `case` example, double interpretation makes the runtime much longer (case 1). After compiling the metalevel interpreter, the execution time is reduced of factor 12. However, it is still slower than the execution time for `(fib2 17)` under the standard semantics (case 2 of Table 7.1).

One of the reasons for this is that the latter is executed under the standard semantics whereas the former is under lazy evaluation. To make fair comparison, we have implemented another version of Black in which the lazy evaluation strategy is employed[1]. Under this modified Black, `(fib 17)` took about 15 seconds, which is still more than twice as fast as `(fib 17)` under our compiled interpreter.

The other reason would be that the lazy evaluation can not be compiled by a simple syntactic dispatch. In the `case` example, `case` statements could be statically dispatched to `eval-case` at partial evaluation time. In the lazy evaluation example, however, arguments are forced according to the value of the operator to be applied. When it is a primitive, arguments are forced. When others, their evaluation is delayed. Since the value of the operator to be applied is unknown until runtime, this means that the entire program to force or make thunk

[1]Notice that this modification is done by directly modifying the implementation of Black, not through `exec-at-metalevel`. Thus, users have to know about the low-level implementation of Black to do this modification.

is residualized in the compiled program. This could be a reason why compiled `fib` (case 3) runs almost the same as efficient as the case 2 of `fib` under the standard semantics. However, further investigation is required here.

Chapter 8

Related Work

8.1 Black Interpreter

Since Black was originally designed to clarify various semantic aspects of Rscheme[30], it is strongly influenced by Rscheme. In both Black and Rscheme, any 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[31] 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[43] 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[27, 48] and later refined in Blond[21]. 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[24, 45] 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[23]. We conjecture that the 3-LISP interpreter is similar to the Black interpreter implemented without using metacontinuations.

8.2 Partial Evaluation

8.2.1 Side-effect Analysis

The side-effect analysis we have used is based on Consel's binding time analysis[19], which introduces the notion of cons points and closure points. We have extended it to return possible values for expressions rather than their binding times. The resulting analysis becomes quite similar to other existing analyses such as a set-based analysis[29]. In fact, the same information can be obtained by using the set-based analysis, which deduces a set of possible values for higher-order λ-calculus with updatable arrays.

The effect analysis by Talpin and Jouvelot[46] is another analysis that gives us similar information. Because the effect analysis handles polymorphic functions, it sometimes produces more accurate information than our analysis. However, in the presence of higher-order functions, the effect analysis requires to unify effect information. This results in more data to be classified as mutable, making the partial evaluation poor.

For C-like programs, there are several analyses which are context-sensitive, e. g., Landi, Ryder, and Zhang[33], Choi, Burke, and Carini[16], and Emami, Ghiya, and Hendren[26]. Ashley[9] showed a flow analysis for higher-order languages with side-effects, which can tune the accuracy of the analysis.

8.2.2 Partial Evaluation of Side-effecting Operations

There are several partial evaluators for Scheme. Among them, Similix[14] is a publicly available offline partial evaluator. Although it can deal with I/O operations and limited types of variable assignments, it does not handle destructive updates of data structures. Schism[20] is also an offline partial evaluator for Scheme, but does not address the partial evaluation of side-effects. POPE[39] is an online partial evaluator for Scheme, which is also publicly available. It accepts richer kinds of variable assignments than Similix, but does not handle destructive updates of data structures correctly.

Fuse[40] is an online partial evaluator, which our partial evaluator was originally based on. It uses graph structure to avoid code duplication. Using preactions, we extended it to handle side-effects, which Fuse does not handle[1]. We also use names to simplify the codification phase considerably.

[1]There seems to be a version of Fuse that handles side-effects[50]. However, actual treatment of side-effects is not clear.

Partial evaluation of side-effects has been considered in imperative languages, such as Pascal[36], Fortran[10], and C[2]. Among them, C-MIX is a powerful partial evaluator that can deal with pointers. It incorporates various analyses (e.g., a pointer analysis and a data-flow analysis) to obtain side-effect information. We performed a similar analysis for higher-order languages where local closures are allowed. Currently, we do not reduce assignments because it does not improve the result very much in functional languages. In imperative languages, most computation is done by assignments to local variables, and thus reducing them is essential. In functional languages, because this type of computation is realized without using assignments, we can still obtain sufficient specializations without reducing them.

Recently, Dussart and Thiemann[25] developed a partial evaluator for ML with a reference type that can statically reduce local static side-effects. In this respect, their partial evaluator is more powerful than ours. Since their specializer is based on Similix's let-expressions, it would be difficult to reduce programs where the preservation of pointer equality is essential, such as Scheme programs.

8.2.3 Preactions

Preactions were originally proposed by Masuhara[34] as a simple mechanism to handle I/O-type side-effects. In this thesis, we have introduced a side-effect analysis to extend the framework to other types of side-effects in a safe way. We have also implemented a realistic partial evaluator for almost full Scheme, and proved its correctness.

Preactions used in this paper have close relationship to the let-expression technique[14, 11] used in Similix[13]. When unknown expressions are encountered in Similix, let-expressions are constructed to residualize them. Since names bound to them are used in the subsequent partial evaluation, problems such as code elimination and code duplication are avoided in the same way as preactions are used. The difference arises when those residualized expressions are *used* in the subsequent partial evaluation. When preactions are used, they are still accessible, enabling further specialization. If let-expressions are used, on the other hand, they become unknown and prohibited from accessing. Remember that λ-expressions and *cons* expressions have side-effects on the store (to declare their identity) and thus residualized as preactions. Treating them unknown results in a very poor specialization. The preaction mechanism enables us to treat expressions as both dynamic and static.

However, it might be possible to use let-expressions in place of preactions, if we residualized them in let-expressions and used them at the same time. We manipulate two data for one expression: its code (or name) and its value. Although it would spoil the efficiency of offline partial evaluators, it would still be possible to make an offline partial evaluator using let-expressions, which we think is interesting future work.

The preaction mechanism has also some similarity to A-normal form[41], which gives names to all subexpression of a program. Code duplication/elimination problem could be avoided by first transforming programs into A-normal form and then using both the names and the values of the subexpression during partial evaluation. This might simplify the correctness proof of our partial evaluator.

In Mogensen's original paper on partially static data structures[37], he mentioned a use of *field* to avoid the code duplication problem. The preaction mechanism can be regarded as

its generalization where residualized expressions are not necessarily dynamic ones.

8.3 Compilation of Reflective Languages

Partial evaluation has not been used to compile reflective languages very much. The only exception is ABCL/R3[35]. ABCL/R3 is a concurrent object-oriented language with the reflective facility. It incorporates partial evaluation to reduce the interpretation overhead of modified interpreters, having quite a similar structure as Black. Because ABCL/R3 has only two levels and modifiable parts are restricted, its sequential parts can be regarded as a special case of Black. We hope that Black forms a basis of the compilation framework of ABCL/R3.

OpenC++[15] is an efficient reflective language based on C++. It uses a compile-time MOP (meta object protocol) framework. To change the semantics in OpenC++, users modify the compiler rather than the interpreter. In Black, users modify the semantics by changing the interpreter $\mathtt{L}_1^{env_1}$. Then, compilation is done by partial evaluation using it: $\mathcal{PE}[\![\mathtt{L}_1^{env_1}(\cdot)]\!]$. In OpenC++, users modify $\mathcal{PE}[\![\mathtt{L}_1^{env_1}(\cdot)]\!]$ directly. This means that users have to know about the actual implementation of the system, which is in contrast to Black where actual implementation is encapsulated by the use of the user-observable interpreter. However, because of this ability to tune a compiler, it would be interesting to incorporate this framework to Black.

Chapter 9

Conclusion and Future Work

In the first part of the thesis, 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.

In the second part of the thesis, we have presented a framework of an online partial evaluator for a call-by-value λ-calculus that deals with destructive updates of data structures and preserves pointer equality. The two key techniques used here are the side-effect analysis and preactions. The side-effect analysis is used to separate immutable cells from mutable ones. Since mutable parts are usually small in functional languages, it enables us to reduce most accesses to structured data at partial evaluation time. For the correct residualization of side-effecting operations, preactions are used to solve various issues, such as code elimination, code duplication, and the order of execution. Partially static data structures are naturally realized by residualizing them as preactions and using them at the same time in the subsequent partial evaluation. This makes more expressions to be reduced at specialization time, which were residualized and treated as dynamic when the conventional let-expression was used. The correctness of the partial evaluator is proven. Based on the framework, we have constructed a partial evaluator for almost full Scheme, which is powerful enough to specialize an interpreter written using side-effects.

In the last part of the thesis, we have actually applied the partial evaluation technique to compile various programs in Black. This can be also regarded as a non-trivial application of partial evaluation. By specializing the current interpreter with respect to a program, it was

compiled under the current language semantics. Furthermore, a user program was compiled under a modified language semantics by first compiling the modified interpreter and then specializing it further with respect to the user program.

As a future research direction, we are mainly thinking about three topics. First, the precision of the side-effect analysis should be improved. The result of partial evaluation is strongly affected by the result of side-effect analysis. We have already found several examples in which side-effect analysis returns poor results, making the partial evaluation almost impossible. By identifying mutable cells more accurately probably in a context-sensitive manner, the quality of compiled code could be improved.

Second, the performance of the partial evaluator itself should be improved. Currently, specialization of compiled programs takes considerable amount of time. If we could incorporate the preaction mechanism to offline partial evaluators, compilation could be done much more efficiently. It would be also interesting to employ the cogen approach[12] in Black rather than the partial evaluation approach. Incorporation of A-normal form[41] should also be investigated.

Third, compilation of more dynamic programs should be considered. In this thesis, we assume that the language semantics does not change during program execution. However, reflective languages are the most interesting and useful when the semantics changes dynamically. To compile such programs, we would have to enumerate possible changes, make compiled code for all of them, and switch them during execution. Or, we would even need dynamic compilation during execution. Although these are still quite difficult, they will certainly make reflective languages more useful and practical.

Bibliography

[1] Abelson, H., and G. J. Sussman with J. Sussman *Structure and Interpretation of Computer Programs*, Cambridge: MIT Press (1985).

[2] Andersen, L. O. *Program Analysis and Specialization for the C Programming Language*, Ph.D. thesis, DIKU, University of Copenhagen (May 1994).

[3] Asai, K., H. Masuhara, S. Matsuoka, and A. Yonezawa "Partial evaluation as a compiler for reflective languages," Technical report of Department of Information Science, University of Tokyo, 95-10 (December 1995).

[4] Asai, K., H. Masuhara, and A. Yonezawa "Partial Evaluation of Call-by-value λ-calculus with Side-effects," Technical report of Department of Information Science, University of Tokyo, 96-04 (November 1996).

[5] Asai, K., H. Masuhara, and A. Yonezawa "Partial Evaluation of Call-by-value Lambda-calculus with Side-effects," to appear in *ACM SIGPLAN Conference on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '97)* (June 1997).

[6] Asai, K., S. Matsuoka, and A. Yonezawa "Duplication and Partial Evaluation to Implement Reflective Languages," *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '93) Workshop on Object-Oriented Reflection and Meta-level Architectures*, Washington D.C. (1993).

[7] Asai, K., S. Matsuoka, and A. Yonezawa "Roles of a Partial Evaluator for the Reflective Language Black," Technical report of Department of Information Science, University of Tokyo, 94-11 (May 1994).

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

[9] Ashley, J. M. "A Practical and Flexible Flow Analysis for Higher-Order Languages," *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pp. 184–194 (January 1996).

[10] Baier, R., R. Glück, and R. Zöchling "Partial Evaluation of Numerical Programs in Fortran," *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '94)*, pp. 119–132 (June 1994).

[11] Bondorf, A. "Automatic autoprojection of higher order recursive equations," *Science of Computer Programming*, Vol. 17, pp. 3–34, Elsevier (1991).

[12] Bondorf, A., and D. Dussart "Improving CPS-Based Partial Evaluation: Writing Cogen by Hand," *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '94)*, pp. 1–9 (June 1994).

[13] Bondorf, A. *Similix 5.0 Manual*, DIKU, University of Copenhagen, (May 1993).

[14] Bondorf, A., and O. Danvy "Automatic autoprojection of recursive equations with global variables and abstract data types," *Science of Computer Programming*, Vol. 16, pp. 151–195, Elsevier (1991).

[15] Chiba, S. "A study of Compile-time Metaobject Protocol," Doctoral thesis of Department of Information Science, University of Tokyo (November 1996).

[16] Choi, J. D., M. Burke, and P. Carini "Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects," *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, pp. 232–245 (January 1993).

[17] Clinger, W., and Rees, J. (editors) "Revised^4 Report on the Algorithmic Language Scheme", *LISP Pointers*, Vol. IV, No. 3, pp. 1–55 (July-September 1991).

[18] Consel, C. "New Insights into Partial Evaluation: the SCHISM Experiment," *ESOP'88, 2nd European Symposium on Programming (LNCS 300)*, pp. 236–246 (March 1988).

[19] Consel, C. "Binding Time Analysis for Higher Order Untyped Functional Languages," *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 264–272 (June 1990).

[20] Consel, C. "A Tour of Schism: A Partial Evaluation System for Higher-Order Applicative Languages," *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pp. 145–154 (June 1993).

[21] 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).

[22] Dean, J., C. Chambers, and D. Grove "Identifying Profitable Specialization in Object-Oriented Languages," *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '94)*, pp. 85–96 (June 1994).

[23] des Rivières, J. private communication, (September 1993).

[24] 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).

[25] Dussart, D., and P. Thiemann "Partial Evaluation for Higher-Order Languages with State," Submitted for publication (1996).

[26] Emami, M., R. Ghiya, and L. J. Hendren "Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers," *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pp. 242–256 (June 1994).

[27] 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).

[28] Futamura, Y. "Partial evaluation of computation process – an approach to a compiler-compiler," *Systems, Computers, Controls*, Vol. 2, No. 5, pp. 45–50, (1971).

[29] Heintze, N. "Set-Based Analysis of ML Programs," *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pp. 306–317 (June 1994).

[30] 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).

[31] 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 appears in *Lisp and Symbolic Computation*, Vol. 9, Nos. 2/3, pp. 181–202, Kluwer Academic Publishers (May/June 1996).

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

[33] Landi, W., B. G. Ryder, and S. Zhang "Interprocedural Modification Side Effect Analysis With Pointer Aliasing," *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, pp. 56–67 (June 1993).

[34] Masuhara, H. et al. "A simple mechanism to handle I/O-type side-effects in on-line partial evaluators," in preparation.

[35] Masuhara, H., S. Matsuoka, K. Asai, and A. Yonezawa "Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation," *Tenth Annual Conference of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, pp. 300–315, (October 1995).

[36] Meyer, U. "Techniques for partial evaluation of imperative languages," *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '91)*, pp. 94–105 (June 1991).

[37] Mogensen, T. Æ. "Partially Static Structures in a Self-Applicable Partial Evaluator," In D. Bjørner, A. P. Ershov, and N. D. Jones editors, *Partial Evaluation and Mixed Computation*, Elsevier Science Publishers, pp. 325–347 (1988).

[38] Mogensen, T. Æ. "Separating Binding Times in Language Specifications," *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pp. 12–25 (September 1989).

[39] Ørbæk, P. "POPE: An On-line Partial Evaluator," available from `ftp://ftp.daimi.aau.dk/pub/empl/poe/pope.ps.gz`, (June 1994).

[40] Ruf, E. *Topics in Online Partial Evaluation*, Ph.D. thesis, Stanford University (March 1993). Also published as Stanford Computer Systems Laboratory technical report CSL-TR-93-563.

[41] Sabry, A., and M. Felleeisen "Reasoning about Programs in Continuation-Passing Style," *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pp. 288–298 (June 1992).

[42] Shivers, O. "Control Flow Analysis in Scheme," *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pp. 164–174 (June 1988).

[43] Simmons II, J. W., S. Jefferson, and D. P. Friedman "Language Extension Via First-class Interpreters," Indiana University Technical Report No. 362, (September 1992).

[44] Sperber, M., and P. Thiemann "The Essence of LR Parsing," *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'95)*, pp. 146–155 (June 1995).

[45] 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).

[46] Talpin, J. P., and P. Jouvelot "Polymorphic type, region and effect inference," *Journal of Functional Programming*, Vol. 2, No. 3, pp. 245–271, Cambridge University Press (July 1992).

[47] Thiemann, P. "Towards Partial Evaluation of Full Scheme," *Proceedings of Reflection'96*, pp. 105–115 (April 1996).

[48] 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).

[49] Weise, D., R. Conybeare, E. Ruf, and S. Seligman "Automatic Online Partial Evaluation," In J. Hughes, editor, *Functional Programming Languages and Computer Architecture (LNCS 523)*, pp. 165–191 (August 1991).

[50] Weise, D., and S. Seligman "Accelerating Object-Oriented Simulation via Automatic Program Specialization," Technical Report of Computer Systems Laboratory, Stanford University, CSL-TR-92-519 (also FUSE Memo 92-10), (April 1992).

Appendix A

Miscellaneous Code

The Appendix A contains the code for Black that did not appear in the main body of the thesis. The code in Chapter 4 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 Chapter 4 (other than Section 4.1) and Appendix A 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? operator)
  (pair? (memq operator primitive-procedures)))
```

```
(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 primitive?
        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))))
```

Appendix B

Black Program for Lazy Evaluation

Appendix B contains a Black program that changes the evaluation strategy of the level below to lazy evaluation.

```
(define thunk-tag (cons 'thunk 'tag))
(define (thunk? x)
  (if (pair? x)
      (eq? (car x) thunk-tag)
      #f))
(define (make-thunk exp env cont)
  (cont (cons thunk-tag
              (let ((result-ready? #f)
                    (result #f))
                (lambda (cont2)
                  (if result-ready?
                     (cont2 result)
                     (base-eval exp env
                                (lambda (ans)
                                  (if result-ready?
                                      (cont2 result)
                                      (begin (set! result-ready? #t)
                                             (set! result ans)
                                             (cont2 result))))))))))))
(define (make-thunk-list lst1 env cont)
  (filter (if (known? lst1) 'unfold '(false false false)))
  (if (null? lst1)
      (cont '())
      (make-thunk (car lst1) env (lambda (val3)
        (make-thunk-list (cdr lst1) env (lambda (val2)
          (cont (cons val3 val2))))))))
(define (force thunk env cont)
  (filter (if (known? thunk) 'unfold '(false false false)))
  (if (thunk? thunk)
      ((cdr thunk)
       (lambda (ans) (force ans env cont)))
      (cont thunk)))
```

```
(define (force-list thunk-list env cont)
  (filter (if (known? thunk-list) 'unfold '(false false false)))
  (if (null? thunk-list)
      (cont '())
      (force (car thunk-list) env
                  (lambda (val4)
                    (force-list (cdr thunk-list) env
                                (lambda (val2)
                                  (cont (cons val4 val2))))))))
(define (eval-application exp env cont)
  (base-eval (car exp) env (lambda (operator)
    (force operator env (lambda (operator)
      (if (primitive? operator)
          (eval-list (cdr exp) env (lambda (operand)
            (force-list operand env (lambda (operand2)
              (base-apply operator operand2 env cont)))))
          (make-thunk-list (cdr exp) env (lambda (operand)
            (base-apply operator operand env cont)))))))))
(define (eval-if exp env cont)
  (let ((pred-part (car (cdr exp)))
        (then-part (car (cdr (cdr exp))))
        (else-part (cdr (cdr (cdr exp)))))
    (base-eval pred-part env (lambda (p)
      (force p env (lambda (p)
        (cond (p (base-eval then-part env cont))
              ((null? else-part) (cont #f))
              (else
               (base-eval (car else-part) env cont)))))))))
(define (init-cont env level turn cont)
  (cont
    (lambda (answer)
      (force answer env (lambda (answer)
        (write level) (write '-) (write turn) (display ": ")
        (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)))))))))))
```

Appendix C

Correctness of the Partial Evaluator

In the Appendix C, we show that the partial evaluator presented in Chapter 5 preserves the semantics of the input program. We first present the store semantics $\mathcal{E}$ to be preserved as well as the partial evaluator $\mathcal{PE}$. For a simpler proof, the partial evaluator we present here is slightly modified from the partial evaluator shown in Chapter 5. Then the correctness proof is given. Because our partial evaluator consists of two parts (specialization $\mathcal{PE}$ and codification $\mathcal{PD}$), the correctness proof is also divided into two parts: the correctness of $\mathcal{PE}$ and $\mathcal{PD}$.

C.1 Store Semantics $\mathcal{E}$

The semantics of the core language (plus *let* expression) is shown in Figure C.1. It is written in the standard store-passing style. (It is based on R^{4}RS[17], but written in the direct style, rather than the continuation passing style.) Given an expression, the semantic function $\mathcal{E}$ evaluates it under an environment ρ and a store σ, and returns a result value together with a possibly modified store.

To make the correctness proof simple, we store all the values of subexpressions of an input program into the store, as shown in the rule for $\mathcal{E}[\![M]\!]\rho\sigma$ (the second rule in Figure C.1). We introduce a fresh hatted name $\hat{\underline{t}}$ and bind it the value of M. The hatted names correspond to the names introduced in preactions. Because hatted names are freshly introduced and will never be accessed afterwards, they can be safely ignored during standard evaluation.

Denotable values in the core language is either a λ-abstraction or a cons cell. They are respectively represented as a triplet of the form $\langle \eta, t, \lambda x \ldots \rangle$ or $\langle \epsilon, t_m, t_n \rangle$, where t, t_m, and t_n are *names* or *locations*.

A function value $\langle \eta, t, \lambda x \ldots \rangle$ indicates that a λ-expression with a closure point η is allocated at the location t. The equality of λ-abstractions are judged by the equality of their allocated locations. Since the name t in $\langle \eta, t, \lambda x \ldots \rangle$ is used only for the equality of functions, the value of t in a store σ is irrelevant and will never be looked up. In the rule for λ-expressions, we explicitly indicate this by binding $\sigma(t)$ to $*$.

A cons cell $\langle \epsilon, t_m, t_n \rangle$ is constructed by a *cons* expression with a cons point ϵ. It holds locations for its *car* and *cdr* part. Two cons cells $\langle \epsilon, t_m, t_n \rangle$ and $\langle \epsilon', t'_m, t'_n \rangle$ are equal when

$$
\begin{array}{rrrl}
 & & \mathcal{E} & : \; Exp \to Env \to Store \to (Value \times Store) \\
\rho & \in & Env & : \; Var \to Name \\
\sigma & \in & Store & : \; Name \to Value \\
 & & Value & : \; Cell \text{ or } Fun \\
 & & Cell & : \; \langle ConsPoint, Name, Name \rangle \\
 & & Fun & : \; \langle ClosurePoint, Name, Arg \to Store \to (Value \times Store) \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![x]\!]\rho\sigma = (\sigma(\rho(x)), \sigma) \\
\mathcal{E}[\![M]\!]\rho\sigma = \textbf{let } (v, \sigma') = \mathcal{E}'[\![M]\!]\rho\sigma \textbf{ in } (v, \sigma'[v/\hat{\underline{t}}])
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}'[\![\eta : \lambda x.M]\!]\rho\sigma = (\langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}[\![M]\!]\rho[\underline{t'}/x]\sigma'[x'/\underline{t'}]\rangle, \sigma[*/\underline{t}]) \\
\mathcal{E}'[\![MN]\!]\rho\sigma = \textbf{let } (\langle \eta, t, m\rangle, \sigma_1) = \mathcal{E}[\![M]\!]\rho\sigma \\
\qquad\qquad\qquad\qquad (n, \sigma_2) = \mathcal{E}[\![N]\!]\rho\sigma_1 \\
\qquad\qquad\qquad \textbf{in } mn\sigma_2 \\
\mathcal{E}'[\![\epsilon : consMN]\!]\rho\sigma = \textbf{let } (m, \sigma_1) = \mathcal{E}[\![M]\!]\rho\sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad (n, \sigma_2) = \mathcal{E}[\![N]\!]\rho\sigma_1 \\
\qquad\qquad\qquad\qquad\qquad \textbf{in } (\langle \epsilon, \underline{t_m}, \underline{t_n}\rangle, \sigma_2[m/\underline{t_m}, n/\underline{t_n}]) \\
\mathcal{E}'[\![carM]\!]\rho\sigma = \textbf{let } (\langle \epsilon, t_m, t_n\rangle, \sigma') = \mathcal{E}[\![M]\!]\rho\sigma \\
\qquad\qquad\qquad\qquad \textbf{in } (\sigma'(t_m), \sigma') \\
\mathcal{E}'[\![setcarMN]\!]\rho\sigma = \textbf{let } (\langle \epsilon, t_m, t_n\rangle, \sigma_1) = \mathcal{E}[\![M]\!]\rho\sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad (n, \sigma_2) = \mathcal{E}[\![N]\!]\rho\sigma_1 \\
\qquad\qquad\qquad\qquad\qquad \textbf{in } (n, \sigma_2[n/t_m]) \\
\mathcal{E}'[\![eqMM'NN']\!]\rho\sigma = \textbf{let } (m, \sigma_1) = \mathcal{E}[\![M]\!]\rho\sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (m', \sigma_2) = \mathcal{E}[\![M']\!]\rho\sigma_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in if } m = m' = \langle \epsilon, t_1, t_2\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then } \mathcal{E}[\![N]\!]\rho\sigma_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else if } m = m' = \langle \eta, t_3, _\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then } \mathcal{E}[\![N]\!]\rho\sigma_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathcal{E}[\![N']\!]\rho\sigma_2
\end{array}
$$

$$
\mathcal{E}[\![let\ x = M\ in\ N]\!]\rho\sigma = \textbf{let } (v, \sigma') = \mathcal{E}'[\![M]\!]\rho\sigma \textbf{ in } \mathcal{E}[\![N]\!]\rho[\underline{t'}/x]\sigma'[v/\underline{t'}]
$$

Figure C.1: Standard Semantics of the Core Language

$t_m = t'_m$ and $t_n = t'_n$. (In this case, ϵ and ϵ' are always the same.)

As was in the rules for $\mathcal{PE}$, names are underlined when they are intended to be fresh names. Fresh names are introduced in three ways (excluding hatted names mentioned above): when a λ-expression is allocated ($\underline{t}$ in the rule for λ-expression), when a λ-abstraction is applied and its bound variable is allocated ($\underline{t'}$ in the rule for λ-expression), and when a cons cell is allocated ($\underline{t_m}$ and $\underline{t_n}$ in the rule for cons cells). We use '_' when we do not care about its value. We say a name is *mutable* if it is updated by *setcar* or *setcdr*. Otherwise, it is *immutable*.

C.2 The Partial Evaluator $\mathcal{PE}$

We re-show the partial evaluator $\mathcal{PE}$ in Figure C.2. It is modified from the original partial evaluator presented in Figure 5.4 in three ways to simplify the correctness proof. First, we renamed fresh names $\underline{t}$ introduced in preactions to fresh hatted names $\hat{\underline{t}}$. These names corresponds to the hatted names in standard semantics. The second change is in the rule for *car*. When $\gamma'(t)$ is a known pair: $\mathsf{pair}(\epsilon, t_1, t_2)$ and $\epsilon \notin \mathit{mutable}_{car}$, we say $\mathbf{let}\ s = \mathsf{car}(t)\ \mathbf{in}\ (^{\langle\langle P, \hat{\underline{t'}}; s\rangle\rangle}t_1, \gamma'[s/\hat{\underline{t'}}])$ instead of $(^{\langle\langle P\rangle\rangle}t_1, \gamma')$. This is because all subexpressions are bound to hatted names in the standard semantics. We can safely ignore $\hat{\underline{t'}} : s$ and $[s/\hat{\underline{t'}}]$, since $\mathsf{car}(t)$ has no side-effects and $\hat{\underline{t'}}$ is fresh, which means that it will never be accessed afterwards. Thirdly, we expand the abbreviation $^{\langle\langle P\rangle\rangle}\mathcal{PE}[\![M]\!]\rho\sigma$ in the rules for applications and *eq*. Here, we introduce hatted names for the same reason mentioned above for *car*. We say $(^{\langle\langle P_1, P_2, \hat{\underline{t}}; s\rangle\rangle}t_3, \gamma_3[s/\hat{\underline{t}}])$ instead of $(^{\langle\langle P_1, P_2, P_3\rangle\rangle}t_3, \gamma_3)$. The semantics of these two are the same except for the allocation of $\hat{\underline{t}}$, because the evaluation of $\hat{\underline{t}} : s$ (where $s = \mathsf{apply}(t_1, t_2)$ for the application rule) makes the same side-effects as P_3.

We note that γ plays a role only when its bound value is a known Sval. The places where γ is used are the rules for applications, *car*, and *eq*. In all cases, γ is used to see if its value is known. When the value is a pair, its component might be accessed. When it is a closure, it might be applied.

C.3 Semantics of PSval $\mathcal{E}_{PS}$ and the Correctness of $\mathcal{PD}$

In this section, we prove the correctness of the codification function $\mathcal{PD}$. The correctness of the specialization function $\mathcal{PE}$ is proved in the next section. The correctness of the codification function $\mathcal{PD}$ is proved with respect to the store semantics $\mathcal{E}$ and the semantics of PSval $\mathcal{E}_{PS}$. Here, we define $\mathcal{E}_{PS}$ in such a way that the semantics of PSval is given by the semantics of its codified program. Figure C.3 shows the definition of $\mathcal{E}_{PS}$, which is quite similar to the store semantics.

Theorem 1 (Correctness of $\mathcal{PD}$) $\mathcal{E}_{PS}[\![^{\langle\langle P\rangle\rangle}t]\!]\sigma = \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P\rangle\rangle}t]\!]]\!]\rho_{id}\sigma$.

Here, ρ_{id} is an identity environment $\lambda t.t$. Because codified programs use freshly introduced names (or locations) as variables, we do not need environments. The theorem says that if $^{\langle\langle P\rangle\rangle}t$ evaluates to v and modifies σ to σ', then its codified program evaluates to the same v and modifies σ to the same σ' (modulo renaming). The proof is by induction on the depth of $\mathcal{PD}$ application required to codify $^{\langle\langle P\rangle\rangle}t$.

$$
\begin{array}{rcrcl}
 & & \mathcal{PE} & : & Exp \to Env \to PEStore \to (PSval \times PEStore) \\
\rho & \in & Env & : & Variable \to Name \\
\gamma & \in & PEStore & : & Name \to Sval
\end{array}
$$

$$
\begin{array}{l}
\mathcal{PE}[\![x]\!]\rho\gamma = (^{\langle\langle\rangle\rangle}\rho(x), \gamma) \\
\mathcal{PE}[\![\eta : \lambda x.M]\!]\rho\gamma = \textbf{let } (^{\langle\langle P\rangle\rangle}t'', \gamma') = \mathcal{PE}[\![M]\!]\rho[\underline{t'}/x]\gamma[\mathsf{var}(\underline{t'})/\underline{t'}] \\
\qquad\qquad\qquad\qquad\quad s = \mathsf{closure}(\eta, x, M, \rho, \underline{t'}, ^{\langle\langle P\rangle\rangle}\, t'') \\
\qquad\qquad\qquad\quad \textbf{in } (^{\langle\langle \underline{t}:s\rangle\rangle}\hat{\underline{t}}, \gamma[s/\hat{\underline{t}}]) \\
\mathcal{PE}[\![MN]\!]\rho\gamma = \textbf{let } (^{\langle\langle P_1\rangle\rangle}t_1, \gamma_1) = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\quad (^{\langle\langle P_2\rangle\rangle}t_2, \gamma_2) = \mathcal{PE}[\![N]\!]\rho\gamma_1 \\
\qquad\qquad\quad \textbf{in case } \gamma_2(t_1) \textbf{ of} \\
\qquad\qquad\qquad \mathsf{closure}(\eta, x, M', \rho', _, _) \;:\; \boxed{\begin{array}{l}\textbf{let } (^{\langle\langle P_3\rangle\rangle}t_3, \gamma_3) = \mathcal{PE}[\![M']\!]\rho'[t_2/x]\gamma_2 \\ \qquad s = \mathsf{apply}(t_1, t_2) \\ \textbf{in } (^{\langle\langle P_1, P_2, \underline{t}:s\rangle\rangle}t_3, \gamma_3[s/\hat{\underline{t}}])\end{array}} \\
\qquad\qquad\qquad \textbf{otherwise} \;:\; \begin{array}{l}\textbf{let } s = \mathsf{apply}(t_1, t_2) \\ \textbf{in } (^{\langle\langle P_1, P_2, \hat{\underline{t}}:s\rangle\rangle}\hat{\underline{t}}, \gamma_2[s/\hat{\underline{t}}])\end{array}
\end{array}
$$

Figure C.2: Modified Partial Evaluator (Part 1)

Proof: By induction on the depth of $\mathcal{PD}$ application.

(Base Case) Since $\mathcal{PD}$ is applied only once, the PSval has the form $^{\langle\langle\rangle\rangle}t$.

$$rhs = \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle\rangle\rangle}t]\!]]\!]\rho_{id}\sigma = \mathcal{E}[\![t]\!]\rho_{id}\sigma = (\sigma(\rho_{id}(t)), \sigma) = (\sigma(t), \sigma) = \mathcal{E}_{PS}[\![^{\langle\langle\rangle\rangle}t]\!]\sigma = lhs$$

(Induction Step) Since $\mathcal{PD}$ is applied more than once, the PSval has a sequence of pre-action whose length is more than one. Let $\langle\langle P\rangle\rangle = \langle\langle t_1 : s_1, P'\rangle\rangle$.

$$
\begin{array}{rcl}
lhs & = & \mathcal{E}_{PS}[\![^{\langle\langle t_1:s_1,P'\rangle\rangle}t]\!]\sigma \\
 & = & \textbf{let } (v', \sigma') = \mathcal{E}_S[\![s_1]\!]\sigma \textbf{ in } \mathcal{E}_{PS}[\![^{\langle\langle P'\rangle\rangle}t]\!]\sigma'[v'/t_1] \\
rhs & = & \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle t_1:s_1,P'\rangle\rangle}t]\!]]\!]\rho_{id}\sigma \\
 & = & \mathcal{E}[\![let\ t_1 = \mathcal{D}[\![s_1]\!]\ in\ \mathcal{PD}[\![^{\langle\langle P'\rangle\rangle}t]\!]]\!]\rho_{id}\sigma \\
 & = & \textbf{let } (v'', \sigma'') = \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma \textbf{ in } \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P'\rangle\rangle}t]\!]]\!]\rho_{id}[\underline{t'}/t_1]\sigma''[v''/\underline{t'}] \\
 & = & \textbf{let } (v'', \sigma'') = \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma \textbf{ in } \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P'\rangle\rangle}t]\!]]\!]\rho_{id}\sigma''[v''/t_1]
\end{array}
$$

Since t_1 is chosen freshly at specialization time, we can take $\underline{t'}$ as t_1, keeping the environment to be ρ_{id}. If we can prove $\mathcal{E}_S[\![s_1]\!]\sigma = \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma$ (and hence $v' = v''$ and $\sigma' = \sigma''$), we are done:

$$
\begin{array}{rcl}
rhs & = & \textbf{let } (v', \sigma') = \mathcal{E}_S[\![s_1]\!]\sigma \textbf{ in } \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P'\rangle\rangle}t]\!]]\!]\rho_{id}\sigma'[v'/t_1] \\
 & = & \textbf{let } (v', \sigma') = \mathcal{E}_S[\![s_1]\!]\sigma \textbf{ in } \mathcal{E}_{PS}[\![^{\langle\langle P'\rangle\rangle}t]\!]\sigma'[v'/t_1] \\
 & = & lhs
\end{array}
$$

$$
\begin{array}{l}
\mathcal{PE}[\![\epsilon : consMN]\!]\rho\gamma = \textbf{let } (^{\langle\!\langle P_1 \rangle\!\rangle} t_1, \gamma_1) = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\qquad\qquad\quad (^{\langle\!\langle P_2 \rangle\!\rangle} t_2, \gamma_2) = \mathcal{PE}[\![N]\!]\rho\gamma_1 \\
\qquad\qquad\qquad\qquad\qquad\quad s = \mathsf{pair}(\epsilon, t_1, t_2) \\
\qquad\qquad\qquad\qquad\qquad \textbf{in } (^{\langle\!\langle P_1,P_2,\underline{\hat{t}}:s \rangle\!\rangle} \underline{\hat{t}}, \gamma_2[s/\underline{\hat{t}}]) \\
\mathcal{PE}[\![carM]\!]\rho\gamma = \textbf{let } (^{\langle\!\langle P \rangle\!\rangle} t, \gamma') = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\qquad \textbf{in case } \gamma'(t) \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad \mathsf{pair}(\epsilon, t_1, t_2) \quad : \quad \textbf{if } \epsilon \in mutable_{car} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then let } s = \mathsf{car}(t) \textbf{ in } (^{\langle\!\langle P,\underline{\hat{t}'}:s \rangle\!\rangle} \underline{\hat{t}'}, \gamma'[s/\underline{\hat{t}'}]) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \boxed{\textbf{let } s = \mathsf{car}(t) \textbf{ in } (^{\langle\!\langle P,\underline{\hat{t}'}:s \rangle\!\rangle} t_1, \gamma'[s/\underline{\hat{t}'}])} \\
\qquad\qquad\qquad\qquad\qquad \textbf{otherwise} \quad : \quad \textbf{let } s = \mathsf{car}(t) \textbf{ in } (^{\langle\!\langle P,\underline{\hat{t}'}:s \rangle\!\rangle} \underline{\hat{t}'}, \gamma'[s/\underline{\hat{t}'}]) \\
\mathcal{PE}[\![setcarMN]\!]\rho\gamma = \textbf{let } (^{\langle\!\langle P_1 \rangle\!\rangle} t_1, \gamma_1) = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\qquad\quad (^{\langle\!\langle P_2 \rangle\!\rangle} t_2, \gamma_2) = \mathcal{PE}[\![N]\!]\rho\gamma_1 \\
\qquad\qquad\qquad\qquad\quad s = \mathsf{setcar}(t_1, t_2) \\
\qquad\qquad\qquad\qquad \textbf{in } (^{\langle\!\langle P_1,P_2,\underline{\hat{t}}:s \rangle\!\rangle} \underline{\hat{t}}, \gamma_2[s/\underline{\hat{t}}]) \\
\mathcal{PE}[\![eqMM'NN']\!]\rho\gamma = \textbf{let } (^{\langle\!\langle P_1 \rangle\!\rangle} t_1, \gamma_1) = \mathcal{PE}[\![M]\!]\rho\gamma \\
\qquad\qquad\qquad\qquad\qquad\quad (^{\langle\!\langle P_2 \rangle\!\rangle} t_2, \gamma_2) = \mathcal{PE}[\![M']\!]\rho\gamma_1 \\
\qquad\qquad\qquad\qquad\quad \textbf{in if } \gamma_2(t_1) \text{ and } \gamma_2(t_2) \text{ are known} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{then if } t_1 = t_2 \textbf{ then } \boxed{\begin{array}{l}\textbf{let } (^{\langle\!\langle P_3 \rangle\!\rangle} t_3, \gamma_3) = \mathcal{PE}[\![N]\!]\rho\gamma_2 \\ \qquad s = \mathsf{eq}(t_1, t_2, {}^{\langle\!\langle P_3 \rangle\!\rangle} t_3, _) \\ \textbf{in } (^{\langle\!\langle P_1,P_2,\underline{\hat{t}}:s \rangle\!\rangle} t_3, \gamma_3[s/\underline{\hat{t}}])\end{array}} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \boxed{\begin{array}{l}\textbf{let } (^{\langle\!\langle P_4 \rangle\!\rangle} t_4, \gamma_4) = \mathcal{PE}[\![N']\!]\rho\gamma_2 \\ \qquad s = \mathsf{eq}(t_1, t_2, _, {}^{\langle\!\langle P_4 \rangle\!\rangle} t_4) \\ \textbf{in } (^{\langle\!\langle P_1,P_2,\underline{\hat{t}}:s \rangle\!\rangle} t_4, \gamma_3[s/\underline{\hat{t}}])\end{array}} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else let } (^{\langle\!\langle P_3 \rangle\!\rangle} t_3, \gamma_3) = \mathcal{PE}[\![N]\!]\rho\gamma_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\; (^{\langle\!\langle P_4 \rangle\!\rangle} t_4, \gamma_4) = \mathcal{PE}[\![N']\!]\rho\gamma_3 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad\; s = \mathsf{eq}(t_1, t_2, {}^{\langle\!\langle P_3 \rangle\!\rangle} t_3, {}^{\langle\!\langle P_4 \rangle\!\rangle} t_4) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in } (^{\langle\!\langle P_1,P_2,\underline{\hat{t}}:s \rangle\!\rangle} \underline{\hat{t}}, \gamma_4[s/\underline{\hat{t}}])
\end{array}
$$

Figure C.2: Modified Partial Evaluator (Part 2)

$$
\begin{array}{l}
\mathcal{E}_{PS} : \mathit{PSval} \to \mathit{Store} \to (\mathit{Value} \times \mathit{Store}) \\
\mathcal{E}_{PS}[\![^{\langle\langle\rangle\rangle} t]\!]\sigma = (\sigma(t), \sigma) \\
\mathcal{E}_{PS}[\![^{\langle\langle t_1 : s_1, \ldots\rangle\rangle} t]\!]\sigma = \textbf{let } (v', \sigma') = \mathcal{E}_S[\![s_1]\!]\sigma \textbf{ in } \mathcal{E}_{PS}[\![^{\langle\langle \ldots\rangle\rangle} t]\!]\sigma'[v'/t_1] \\
\\
\mathcal{E}_S : \mathit{Sval} \to \mathit{Store} \to (\mathit{Value} \times \mathit{Store}) \\
\mathcal{E}_S[\![\mathsf{closure}(\eta, _, _, _, t', ^{\langle\langle P\rangle\rangle} t'')]\!]\sigma = (\langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}_{PS}[\![^{\langle\langle P\rangle\rangle} t'']\!]\sigma'[x'/t']\rangle, \sigma[*/\underline{t}]) \\
\mathcal{E}_S[\![\mathsf{apply}(t_1, t_2)]\!]\sigma = \textbf{let } \langle \eta, t, m\rangle = \sigma(t_1) \textbf{ in } m\sigma(t_2)\sigma \\
\mathcal{E}_S[\![\mathsf{pair}(\epsilon, t_1, t_2)]\!]\sigma = (\langle \epsilon, \underline{t_m}, \underline{t_n}\rangle, \sigma[\sigma(t_1)/\underline{t_m}, \sigma(t_2)/\underline{t_n}]) \\
\mathcal{E}_S[\![\mathsf{car}(t)]\!]\sigma = \textbf{let } \langle \epsilon, t_m, t_n\rangle = \sigma(t) \textbf{ in } (\sigma(t_m), \sigma) \\
\mathcal{E}_S[\![\mathsf{setcar}(t_1, t_2)]\!]\sigma = \textbf{let } \langle \epsilon, t_m, t_n\rangle = \sigma(t_1) \textbf{ in } (\sigma(t_2), \sigma[\sigma(t_2)/t_m]) \\
\mathcal{E}_S[\![\mathsf{eq}(t_1, t_2, ^{\langle\langle P_3\rangle\rangle} t_3, ^{\langle\langle P_4\rangle\rangle} t_4)]\!]\sigma = \textbf{if } \sigma(t_1) = \sigma(t_2) = \langle \epsilon, t_m, t_n\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then } \mathcal{E}_{PS}[\![^{\langle\langle P_3\rangle\rangle} t_3]\!]\sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else if } \sigma(t_1) = \sigma(t_2) = \langle \eta, t', _\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{then } \mathcal{E}_{PS}[\![^{\langle\langle P_3\rangle\rangle} t_3]\!]\sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathcal{E}_{PS}[\![^{\langle\langle P_4\rangle\rangle} t_4]\!]\sigma
\end{array}
$$

Figure C.3: Semantics of PSval $\mathcal{E}_{PS}$

using the induction hypothesis. Thus, we now prove $\mathcal{E}_S[\![s_1]\!]\sigma = \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma$ by the case analysis of s_1.

(Case for $s_1 = \mathsf{closure}(\eta, _, _, _, t', ^{\langle\langle P''\rangle\rangle} t'')$**)**

$$
\begin{array}{rcl}
\mathit{lhs} & = & \mathcal{E}_S[\![s_1]\!]\sigma \\
& = & (\langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}_{PS}[\![^{\langle\langle P''\rangle\rangle} t'']\!]\sigma'[x'/t']\rangle, \sigma[*/\underline{t}]) \\
\mathit{rhs} & = & \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma \\
& = & \mathcal{E}'[\![\eta : \lambda t'.\mathcal{PD}[\![^{\langle\langle P''\rangle\rangle} t'']\!]]\!]\rho_{id}\sigma \\
& = & (\langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P''\rangle\rangle} t'']\!]]\!]\rho_{id}[\underline{t'}/t']\sigma'[x'/\underline{t'}]\rangle, \sigma[*/\underline{t}]) \\
& = & (\langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P''\rangle\rangle} t'']\!]]\!]\rho_{id}\sigma'[x'/t']\rangle, \sigma[*/\underline{t}])
\end{array}
$$

Here, we take the same $\underline{t}$ as we took for *lhs*. We also choose $\underline{t'}$ to be t' to keep the environment ρ_{id}. From the induction hypothesis, $\mathcal{E}_{PS}[\![^{\langle\langle P''\rangle\rangle} t'']\!]\sigma'[x'/t'] = \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P''\rangle\rangle} t'']\!]]\!]\rho_{id}\sigma'[x'/t']$, and thus $\mathit{lhs} = \mathit{rhs}$.

(Case for $s_1 = \mathsf{apply}(t'_1, t'_2)$**)**

$$
\begin{array}{rcl}
\mathit{lhs} & = & \mathcal{E}_S[\![s_1]\!]\sigma \\
& = & \textbf{let } \langle \eta, t, m\rangle = \sigma(t'_1) \textbf{ in } m\sigma(t'_2)\sigma \\
\mathit{rhs} & = & \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma \\
& = & \mathcal{E}'[\![t'_1 t'_2]\!]\rho_{id}\sigma \\
& = & \textbf{let } (\langle \eta, t, m\rangle, \sigma_1) = \mathcal{E}[\![t'_1]\!]\rho_{id}\sigma \\
& & \quad\; (n, \sigma_2) = \mathcal{E}[\![t'_2]\!]\rho_{id}\sigma_1 \\
& & \textbf{in } mn\sigma_2
\end{array}
$$

Because both t'_1 and t'_2 are variables, we have $\mathcal{E}[\![t'_1]\!]\rho_{id}\sigma = (\sigma(t'_1), \sigma)$, $\mathcal{E}[\![t'_2]\!]\rho_{id}\sigma = (\sigma(t'_2), \sigma)$, $\sigma = \sigma_1 = \sigma_2$, and $n = \sigma(t'_2)$. Thus, $lhs = rhs$.

(Case for $s_1 = \mathsf{pair}(\epsilon, t'_1, t'_2)$)

$$\begin{array}{rcl} lhs & = & \mathcal{E}_S[\![s_1]\!]\sigma \\ & = & (\langle \epsilon, t_m, t_n \rangle, \sigma[\sigma(t'_1)/\underline{t_m}, \sigma(t'_2)/\underline{t_n}]) \\ rhs & = & \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma \\ & = & \mathcal{E}'[\![\epsilon : cons\, t'_1 t'_2]\!]\rho_{id}\sigma \\ & = & \textbf{let } (m, \sigma_1) = \mathcal{E}[\![t'_1]\!]\rho_{id}\sigma \\ & & \quad (n, \sigma_2) = \mathcal{E}[\![t'_2]\!]\rho_{id}\sigma_1 \\ & & \textbf{in } (\langle \epsilon, t_m, t_n \rangle, \sigma_2[m/\underline{t_m}, n/\underline{t_n}]) \end{array}$$

Here, we take the same $\underline{t_m}$ and $\underline{t_n}$ as we took for lhs. Because both t'_1 and t'_2 are variables, we have $\mathcal{E}[\![t'_1]\!]\rho_{id}\sigma = (\sigma(t'_1), \sigma)$, $\mathcal{E}[\![t'_2]\!]\rho_{id}\sigma = (\sigma(t'_2), \sigma)$, $\sigma = \sigma_1 = \sigma_2$, $m = \sigma(t'_1)$, and $n = \sigma(t'_2)$. Thus, $lhs = rhs$.

(Case for $s_1 = \mathsf{car}(t')$)

$$\begin{array}{rcl} lhs & = & \mathcal{E}_S[\![s_1]\!]\sigma \\ & = & \textbf{let } \langle \epsilon, t_m, t_n \rangle = \sigma(t') \textbf{ in } (\sigma(t_m), \sigma) \\ rhs & = & \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma \\ & = & \mathcal{E}'[\![car\, t']\!]\rho_{id}\sigma \\ & = & \textbf{let } (\langle \epsilon, t_m, t_n \rangle, \sigma') = \mathcal{E}[\![t']\!]\rho_{id}\sigma \textbf{ in } (\sigma'(t_m), \sigma') \end{array}$$

Because t' is a variable, we have $\mathcal{E}[\![t']\!]\rho_{id}\sigma = (\sigma(t'), \sigma)$ and $\sigma' = \sigma$. Thus, $lhs = rhs$.

(Case for $s_1 = \mathsf{setcar}(t'_1, t'_2)$)

$$\begin{array}{rcl} lhs & = & \mathcal{E}_S[\![s_1]\!]\sigma \\ & = & \textbf{let } \langle \epsilon, t_m, t_n \rangle = \sigma(t'_1) \textbf{ in } (\sigma(t'_2), \sigma[\sigma(t'_2)/t_m]) \\ rhs & = & \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma \\ & = & \mathcal{E}'[\![setcar\, t'_1 t'_2]\!]\rho_{id}\sigma \\ & = & \textbf{let } (\langle \epsilon, t_m, t_n \rangle, \sigma_1) = \mathcal{E}[\![t'_1]\!]\rho_{id}\sigma \\ & & \quad (n, \sigma_2) = \mathcal{E}[\![t'_2]\!]\rho_{id}\sigma_1 \\ & & \textbf{in } (n, \sigma_2[n/t_m]) \end{array}$$

Because both t'_1 and t'_2 are variables, we have $\mathcal{E}[\![t'_1]\!]\rho_{id}\sigma = (\sigma(t'_1), \sigma)$, $\mathcal{E}[\![t'_2]\!]\rho_{id}\sigma = (\sigma(t'_2), \sigma)$, $\sigma = \sigma_1 = \sigma_2$, and $n = \sigma(t'_2)$. Thus, $lhs = rhs$.

(Case for $s_1 = \mathsf{eq}(t'_1, t'_2, {}^{\langle\langle P'_3 \rangle\rangle}t'_3, {}^{\langle\langle P'_4 \rangle\rangle}t'_4)$)

$$lhs \;=\; \mathcal{E}_S[\![s_1]\!]\sigma$$

$$
\begin{array}{rcl}
 & = & \textbf{if } \sigma(t_1') = \sigma(t_2') = \langle \epsilon, t_m, t_n \rangle \\
 & & \textbf{then } \mathcal{E}_{PS}[\![^{\langle\langle P_3' \rangle\rangle} t_3']\!]\sigma \\
 & & \textbf{else if } \sigma(t_1') = \sigma(t_2') = \langle \eta, t', _ \rangle \\
 & & \quad \textbf{then } \mathcal{E}_{PS}[\![^{\langle\langle P_3' \rangle\rangle} t_3']\!]\sigma \\
 & & \quad \textbf{else } \mathcal{E}_{PS}[\![^{\langle\langle P_4' \rangle\rangle} t_4']\!]\sigma \\
rhs & = & \mathcal{E}'[\![\mathcal{D}[\![s_1]\!]]\!]\rho_{id}\sigma \\
 & = & \mathcal{E}'[\![eq\ t_1'\ t_2'\ \mathcal{PD}[\![^{\langle\langle P_3' \rangle\rangle} t_3']\!]\ \mathcal{PD}[\![^{\langle\langle P_3' \rangle\rangle} t_3']\!]]\!]\rho_{id}\sigma \\
 & = & \textbf{let } (m, \sigma_1) = \mathcal{E}[\![t_1']\!]\rho_{id}\sigma \\
 & & \quad (m', \sigma_2) = \mathcal{E}[\![t_2']\!]\rho_{id}\sigma_1 \\
 & & \textbf{in if } m = m' = \langle \epsilon, t_m, t_n \rangle \\
 & & \quad \textbf{then } \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P_3' \rangle\rangle} t_3']\!]]\!]\rho_{id}\sigma_2 \\
 & & \quad \textbf{else if } m = m' = \langle \eta, t', _ \rangle \\
 & & \quad\quad \textbf{then } \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P_3' \rangle\rangle} t_3']\!]]\!]\rho_{id}\sigma_2 \\
 & & \quad\quad \textbf{else } \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P_4' \rangle\rangle} t_4']\!]]\!]\rho_{id}\sigma_2
\end{array}
$$

Because both t_1' and t_2' are variables, we have $\mathcal{E}[\![t_1']\!]\rho_{id}\sigma = (\sigma(t_1'), \sigma)$, $\mathcal{E}[\![t_2']\!]\rho_{id}\sigma = (\sigma(t_2'), \sigma)$, $\sigma = \sigma_1 = \sigma_2$, $m = \sigma(t_1')$, and $m' = \sigma(t_2')$. From the induction hypotheses, we have $\mathcal{E}_{PS}[\![^{\langle\langle P_3' \rangle\rangle} t_3']\!]\sigma = \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P_3' \rangle\rangle} t_3']\!]]\!]\rho_{id}\sigma$ and $\mathcal{E}_{PS}[\![^{\langle\langle P_4' \rangle\rangle} t_4']\!]\sigma = \mathcal{E}[\![\mathcal{PD}[\![^{\langle\langle P_4' \rangle\rangle} t_4']\!]]\!]\rho_{id}\sigma$. Thus, *lhs* = *rhs*. □

C.4 Correctness of the Partial Evaluator $\mathcal{PE}$

We now prove the correctness of the partial evaluator $\mathcal{PE}$. To prove the correctness of $\mathcal{PE}$, we need three lemmas. The first lemma gives us the way to obtain the semantics of composite PSval from the semantics of component PSvals. The proof is almost trivial.

Lemma 1 *If* $\mathcal{E}_{PS}[\![^{\langle\langle P_1 \rangle\rangle} t_1]\!]\sigma = (v_1, \sigma_1)$ *and* $\mathcal{E}_{PS}[\![^{\langle\langle P_2 \rangle\rangle} t_2]\!]\sigma_1 = (v_2, \sigma_2)$, *then* $\mathcal{E}_{PS}[\![^{\langle\langle P_1, P_2 \rangle\rangle} t_2]\!]\sigma = (v_2, \sigma_2)$.

Proof: By induction on the length of P_1.

(Base Case) P_1 is a null sequence. From

$$(v_1, \sigma_1) = \mathcal{E}_{PS}[\![^{\langle\langle P_1 \rangle\rangle} t_1]\!]\sigma = \mathcal{E}_{PS}[\![^{\langle\langle\rangle\rangle} t_1]\!]\sigma = (\sigma(t_1), \sigma),$$

we have $\sigma_1 = \sigma$ and $v_1 = \sigma(t_1)$. Then,

$$\mathcal{E}_{PS}[\![^{\langle\langle P_1, P_2 \rangle\rangle} t_2]\!]\sigma = \mathcal{E}_{PS}[\![^{\langle\langle P_2 \rangle\rangle} t_2]\!]\sigma = \mathcal{E}_{PS}[\![^{\langle\langle P_2 \rangle\rangle} t_2]\!]\sigma_1 = (v_2, \sigma_2).$$

(Induction Step) P_1 is a sequence of at least length one. Let $\langle\langle P_1 \rangle\rangle = \langle\langle t : s, P_1' \rangle\rangle$. Assuming that $(v', \sigma') = \mathcal{E}_S[\![t : s]\!]\sigma$,

$$(v_1, \sigma_1) = \mathcal{E}_{PS}[\![^{\langle\langle P_1 \rangle\rangle} t_1]\!]\sigma = \mathcal{E}_{PS}[\![^{\langle\langle t:s, P_1' \rangle\rangle} t_1]\!]\sigma = \mathcal{E}_{PS}[\![^{\langle\langle P_1' \rangle\rangle} t_1]\!]\sigma'[v'/t].$$

Since $\mathcal{E}_{PS}[\![^{\langle\langle P_2\rangle\rangle}t_2]\!]\sigma_1 = (v_2, \sigma_2)$, we have $\mathcal{E}_{PS}[\![^{\langle\langle P_1', P_2\rangle\rangle}t_2]\!]\sigma'[v'/t] = (v_2, \sigma_2)$ from the induction hypothesis. Then,

$$\mathcal{E}_{PS}[\![^{\langle\langle P_1, P_2\rangle\rangle}t_2]\!]\sigma = \mathcal{E}_{PS}[\![^{\langle\langle t:s, P_1', P_2\rangle\rangle}t_2]\!]\sigma = \mathcal{E}_{PS}[\![^{\langle\langle P_1', P_2\rangle\rangle}t_2]\!]\sigma'[v'/t] = (v_2, \sigma_2).$$

□

The second lemma is counter-intuitive in the presence of side-effects:

Lemma 2 *The names that appear in a PSval are immutable.*

Because accesses to mutable cells are forbidden, mutable names cannot appear in PSval.

Proof: The name t in a PSval $^{\langle\langle P\rangle\rangle}t'$ is mutable if t is t_m and is updated in the $\mathcal{E}_S$ rule for **setcar** (or **setcdr**) during the evaluation of $\mathcal{E}_{PS}[\![^{\langle\langle P\rangle\rangle}t']\!]\sigma$. Because t_m in the rule for **setcar** can be created only in the rule for **pair** freshly, the only possibility that the mutable name t_m appears in a PSval is by accessing it during partial evaluation. However, in the $\mathcal{PE}$ rule for *car*, pairs are accessed only when their *car* parts are immutable, and thus mutable names cannot appear in PSvals. □

The last lemma relates names to its values:

Lemma 3 *If* $\mathcal{E}_{PS}[\![^{\langle\langle P\rangle\rangle}t]\!]\sigma = (v, \sigma')$, *then* $\sigma'(t) = v$.

Proof: Formally, the proof is by induction on the length of P. However, we can easily see that the evaluation of $\mathcal{E}_{PS}[\![^{\langle\langle P\rangle\rangle}t]\!]\sigma$ will eventually encounter $\mathcal{E}_{PS}[\![^{\langle\langle\rangle\rangle}t]\!]\sigma'$, which returns $(\sigma'(t), \sigma')$. Thus, $\sigma'(t) = v$. □

To relate a store σ and a pe-store γ, we introduce a relation 'respect' between them.

Definition 1 *A pe-store* γ respects *a store* σ, *if the following holds for all* $t \in domain(\gamma)$.

- *if* $\gamma(t) = \mathsf{pair}(\epsilon, t_1, t_2)$, *then* $\sigma(t) = \langle\epsilon, t_m, t_n\rangle$, $\sigma(t_1) = \sigma(t_m)$, *and* $\sigma(t_2) = \sigma(t_n)$, *or* $\sigma(t)$ *is undefined.*
- *if* $\gamma(t) = \mathsf{closure}(\eta, x, M, \rho, _, _)$, *then* $\sigma(t) = \langle\eta, t'', \lambda x'.\lambda\sigma'.\mathcal{E}[\![M]\!]\rho[t'/x]\sigma'[x'/t']\rangle$, *or* $\sigma(t)$ *is undefined.*

The relation 'respect' only requires that if $\gamma(t)$ is known, then $\sigma(t)$ has the same value. Since γ plays a role only when its bound value is known, we do not need to mention the other cases where $\gamma(t)$ is unknown. The reason for having the phrase "or $\sigma(t)$ is undefined" is that we have conditionals. Because the partial evaluator eagerly evaluates both branches of conditionals, some names are defined only in γ and not in σ.

We now show the correctness of $\mathcal{PE}$.

Theorem 2 (Correctness of $\mathcal{PE}$) *If* γ *respects* σ, $\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M]\!]\rho\gamma]\!]\sigma = \mathcal{E}[\![M]\!]\rho\sigma$.

Here, $\mathcal{E}'_{PS}$ takes the PSval part of the $\mathcal{PE}$'s return value: $\mathcal{E}'_{PS}[\![(^{\langle\langle P\rangle\rangle}t, \gamma)]\!]\sigma = \mathcal{E}_{PS}[\![^{\langle\langle P\rangle\rangle}t]\!]\sigma$. The theorem says that if M evaluates under ρ and σ to v and modifies σ to σ', then the result of partially evaluating M under the same ρ and γ evaluates to the same v and modifies σ to the same σ' (modulo renaming). The proof is by induction on the depth of $\mathcal{PE}$ application required to partially evaluate M.

Proof: By induction on the depth of $\mathcal{PE}$ application. In the proof, we prove simultaneously the following: if $\mathcal{PE}[\![M]\!]\rho\gamma = ({}^{\langle\langle P\rangle\rangle}t, \gamma')$ and $\mathcal{E}[\![M]\!]\rho\sigma = (v, \sigma')$, then γ' respects σ'.

(Base Case) Since $\mathcal{PE}$ is applied only once, M is a variable. Assume that γ respects σ.

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![x]\!]\rho\gamma]\!]\sigma = \mathcal{E}'_{PS}[\![({}^{\langle\langle\rangle\rangle}\rho(x), \gamma)]\!]\sigma = \mathcal{E}_{PS}[\![{}^{\langle\langle\rangle\rangle}\rho(x)]\!]\sigma = (\sigma(\rho(x)), \sigma) = \mathcal{E}[\![x]\!]\rho\sigma$$

As $\gamma' = \gamma$ and $\sigma' = \sigma$, γ' respects σ'.

(Induction Step) Case analysis on the form of M. It is a good idea to read from the case for $\epsilon : consMN$, since it gives a typical proof.

(Case for $\eta : \lambda x.M$)

$$\begin{aligned}
rhs &= \mathcal{E}[\![\eta : \lambda x.M]\!]\rho\sigma \\
&= \textbf{let } (v, \sigma') = \mathcal{E}'[\![\eta : \lambda x.M]\!]\rho\sigma \textbf{ in } (v, \sigma'[v/\hat{\underline{t}}]) \\
&= \textbf{let } (v, \sigma') = (\langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}[\![M]\!]\rho[\underline{t'}/x]\sigma'[x'/\underline{t'}]\rangle, \sigma[*/\underline{t}]) \textbf{ in } (v, \sigma'[v/\hat{\underline{t}}]) \\
&= \textbf{let } v = \langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}[\![M]\!]\rho[\underline{t'}/x]\sigma'[x'/\underline{t'}]\rangle \textbf{ in } (v, \sigma[*/\underline{t}, v/\hat{\underline{t}}])
\end{aligned}$$

Let $\mathcal{E}[\![M]\!]\rho[\underline{t'}/x]\sigma'[x'/\underline{t'}] = (m, \sigma_1)$ and $\mathcal{PE}[\![M]\!]\rho[\underline{t'}/x]\gamma[\mathsf{var}(\underline{t'})/\underline{t'}] = ({}^{\langle\langle P\rangle\rangle}t'', \gamma_1)$. Because σ' is a store at the application of this closure, it is an extension of σ. From lemma 2, we have $\sigma(t) = \sigma'(t)$ for all $t \in domain(\gamma)$, and thus, $\gamma[\mathsf{var}(\underline{t'})/\underline{t'}]$ respects $\sigma'[x'/\underline{t'}]$. From the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M]\!]\rho[\underline{t'}/x]\gamma[\mathsf{var}(\underline{t'})/\underline{t'}]]\!]\sigma'[x'/\underline{t'}] = \mathcal{E}[\![M]\!]\rho[\underline{t'}/x]\sigma'[x'/\underline{t'}]$$

we have $\mathcal{E}_{PS}[\![{}^{\langle\langle P\rangle\rangle}t'']\!]\sigma'[x'/\underline{t'}] = (m, \sigma_1)$ and γ_1 respects σ_1. From lemma 3, we have $\sigma_1(t'') = m$.

$$\begin{aligned}
lhs &= \mathcal{E}'_{PS}[\![\mathcal{PE}[\![\eta : \lambda x.M]\!]\rho\gamma]\!]\sigma \\
&= \mathcal{E}_{PS}[\![{}^{\langle\langle \hat{\underline{t}}:\mathsf{closure}(\eta,x,M,\rho,\underline{t'},{}^{\langle\langle P\rangle\rangle}t'')\rangle\rangle}\hat{\underline{t}}]\!]\sigma \\
&= \textbf{let } (v, \sigma') = \mathcal{E}_S[\![\mathsf{closure}(\eta, x, M, \rho, \underline{t'}, {}^{\langle\langle P\rangle\rangle}t'')]\!]\sigma \textbf{ in } \mathcal{E}_{PS}[\![{}^{\langle\langle\rangle\rangle}\hat{\underline{t}}]\!]\sigma'[v/\hat{\underline{t}}] \\
&= \textbf{let } (v, \sigma') = (\langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}_{PS}[\![{}^{\langle\langle P\rangle\rangle}t'']\!]\sigma'[x'/\underline{t'}]\rangle, \sigma[*/\underline{t}]) \textbf{ in } (v, \sigma'[v/\hat{\underline{t}}]) \\
&= \textbf{let } v = \langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}_{PS}[\![{}^{\langle\langle P\rangle\rangle}t'']\!]\sigma'[x'/\underline{t'}]\rangle \textbf{ in } (v, \sigma[*/\underline{t}, v/\hat{\underline{t}}])
\end{aligned}$$

where we take the same $\underline{t}$ and $\hat{\underline{t}}$ as chosen in *rhs*. Using the induction hypothesis, we have $lhs = rhs$.

The pe-store returned form $\mathcal{PE}[\![\eta : \lambda x.M]\!]\rho\gamma$ is $\gamma[\mathsf{closure}(\eta, x, M, \rho, \underline{t'}, {}^{\langle\langle P\rangle\rangle}t'')/\hat{\underline{t}}]$. It respects $\sigma[*/\underline{t}, v/\hat{\underline{t}}] = \sigma[*/\underline{t}, \langle \eta, \underline{t}, \lambda x'.\lambda\sigma'.\mathcal{E}_{PS}[\![{}^{\langle\langle P\rangle\rangle}t'']\!]\sigma'[x'/\underline{t'}]\rangle/\hat{\underline{t}}]$, since γ respects σ and

$$\mathcal{E}_{PS}[\![{}^{\langle\langle P\rangle\rangle}t'']\!]\sigma'[x'/\underline{t'}] = \mathcal{E}'_{PS}[\![\mathcal{PE}[\![M]\!]\rho[\underline{t'}/x]\gamma[\mathsf{var}(\underline{t'})/\underline{t'}]]\!]\sigma'[x'/\underline{t'}] = \mathcal{E}[\![M]\!]\rho[\underline{t'}/x]\sigma'[x'/\underline{t'}]$$

from the induction hypothesis.

(Case for MN) Let $\mathcal{E}[\![M]\!]\rho\sigma = (\langle \eta, t, m\rangle, \sigma_1)$ and $\mathcal{PE}[\![M]\!]\rho\gamma = ({}^{\langle\!\langle P_1 \rangle\!\rangle}t_1, \gamma_1)$. Assume that γ respects σ. From the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M]\!]\rho\gamma]\!]\sigma = \mathcal{E}[\![M]\!]\rho\sigma$$

we have $\mathcal{E}_{PS}[\![{}^{\langle\!\langle P_1 \rangle\!\rangle}t_1]\!]\sigma = (\langle \eta, t, m\rangle, \sigma_1)$ and γ_1 respects σ_1. From lemma 3, we have $\sigma_1(t_1) = \langle \eta, t, m\rangle$.

In the same way, let $\mathcal{E}[\![N]\!]\rho\sigma_1 = (n, \sigma_2)$ and $\mathcal{PE}[\![N]\!]\rho\gamma_1 = ({}^{\langle\!\langle P_2 \rangle\!\rangle}t_2, \gamma_2)$. Since γ_1 respects σ_1, from the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![N]\!]\rho\gamma_1]\!]\sigma_1 = \mathcal{E}[\![N]\!]\rho\sigma_1$$

we have $\mathcal{E}_{PS}[\![{}^{\langle\!\langle P_2 \rangle\!\rangle}t_2]\!]\sigma_1 = (n, \sigma_2)$ and γ_2 respects σ_2. From lemma 3, we have $\sigma_2(t_2) = n$.

$$\begin{aligned} rhs &= \mathcal{E}[\![MN]\!]\rho\sigma \\ &= \textbf{let } (v, \sigma') = \mathcal{E}'[\![MN]\!]\rho\sigma \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\ &= \textbf{let } (v, \sigma') = mn\sigma_2 \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \end{aligned}$$

Since t_1 is fresh and not defined in P_2, we have $\sigma_2(t_1) = \sigma_1(t_1) = \langle \eta, t, m\rangle$. Since γ_2 respects σ_2, if $\gamma_2(t_1)$ has the form: $\mathsf{closure}(\eta, x, M', \rho', _, _)$, this means that $m = \lambda x'.\lambda\sigma'.\mathcal{E}[\![M']\!]\rho'[\underline{t'}/x]\sigma'[x'/\underline{t'}]$.

$$\begin{aligned} rhs &= \textbf{let } (v, \sigma') = mn\sigma_2 \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\ &= \textbf{let } (v, \sigma') = \mathcal{E}[\![M']\!]\rho'[\underline{t'}/x]\sigma_2[n/\underline{t'}] \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \end{aligned}$$

In this case, we use another induction hypothesis. Let $\mathcal{E}[\![M']\!]\rho'[\underline{t'}/x]\sigma_2[n/\underline{t'}] = (m', \sigma_3)$ and $\mathcal{PE}[\![M']\!]\rho'[\underline{t'}/x]\gamma_2 = ({}^{\langle\!\langle P_3 \rangle\!\rangle}t_3, \gamma_3)$. Since γ_2 respects $\sigma_2[n/\underline{t'}]$, from the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M']\!]\rho'[\underline{t'}/x]\gamma[n/\underline{t'}]]\!]\sigma_2[n/\underline{t'}] = \mathcal{E}[\![M']\!]\rho'[\underline{t'}/x]\sigma_2[n/\underline{t'}]$$

$\mathcal{E}_{PS}[\![{}^{\langle\!\langle P_3 \rangle\!\rangle}t_3]\!]\sigma_3 = (m', \sigma_3)$ and γ_3 respects σ_3. From lemma 3, we have $\sigma_3(t_3) = m'$.

$$\begin{aligned} rhs &= \textbf{let } (v, \sigma') = \mathcal{E}[\![M']\!]\rho'[\underline{t'}/x]\sigma_2[n/\underline{t'}] \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\ &= \textbf{let } (v, \sigma') = (m', \sigma_3) \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\ &= (m', \sigma_3[m'/\underline{\hat{t}}]) \\ lhs &= \mathcal{E}'_{PS}[\![\mathcal{PE}[\![MN]\!]\rho\gamma]\!]\sigma \\ &= \mathcal{E}_{PS}[\![{}^{\langle\!\langle P_1, P_2, \underline{\hat{t}}:\mathsf{apply}(t_1,t_2)\rangle\!\rangle}t_3]\!]\sigma \\ \mathcal{E}_{PS}[\![{}^{\langle\!\langle \underline{\hat{t}}:\mathsf{apply}(t_1,t_2)\rangle\!\rangle}t_3]\!]\sigma_2 &= \textbf{let } (v, \sigma') = \mathcal{E}_S[\![\mathsf{apply}(t_1, t_2)]\!]\sigma_2 \textbf{ in } \mathcal{E}_{PS}[\![{}^{\langle\!\langle\rangle\!\rangle}t_3]\!]\sigma'[v/\underline{\hat{t}}] \\ &= \textbf{let } (v, \sigma') = m\sigma_2(t_2)\sigma_2 \textbf{ in } (\sigma'[v/\underline{\hat{t}}](t_3), \sigma'[v/\underline{\hat{t}}]) \\ &= \textbf{let } (v, \sigma') = mn\sigma_2 \textbf{ in } (\sigma'[v/\underline{\hat{t}}](t_3), \sigma'[v/\underline{\hat{t}}]) \\ &= \textbf{let } (v, \sigma') = (m', \sigma_3) \textbf{ in } (\sigma'[v/\underline{\hat{t}}](t_3), \sigma'[v/\underline{\hat{t}}]) \\ &= (\sigma_3[m'/\underline{\hat{t}}](t_3), \sigma_3[m'/\underline{\hat{t}}]) \\ &= (m', \sigma_3[m'/\underline{\hat{t}}]) \end{aligned}$$

where we take the same $\underline{\hat{t}}$ as chosen in *rhs*. Using lemma 1 twice, we have $lhs = rhs$.

The pe-store returned form $\mathcal{PE}[\![MN]\!]\rho\gamma$ in this case is $\gamma_3[\mathsf{apply}(t_1,t_2)/\hat{\underline{t}}]$. It respects $\sigma_3[m'/\hat{\underline{t}}]$, since γ_3 respects σ_3.

If $\gamma_2(t_1)$ is not a **closure** Sval, it means that $\gamma_2(t_1)$ is unknown code. In this case,

$$\begin{array}{rcl} \mathit{lhs} & = & \mathcal{E}'_{PS}[\![\mathcal{PE}[\![MN]\!]\rho\gamma]\!]\sigma \\ & = & \mathcal{E}_{PS}[\![^{\langle\!\langle P_1,P_2,\underline{t}:\mathsf{apply}(t_1,t_2)\rangle\!\rangle}\hat{\underline{t}}]\!]\sigma \\ \mathcal{E}_{PS}[\![^{\langle\!\langle \underline{t}:\mathsf{apply}(t_1,t_2)\rangle\!\rangle}\hat{\underline{t}}]\!]\sigma_2 & = & \textbf{let } (v,\sigma') = \mathcal{E}_S[\![\mathsf{apply}(t_1,t_2)]\!]\sigma_2 \textbf{ in } \mathcal{E}_{PS}[\![^{\langle\!\langle\rangle\!\rangle}\hat{\underline{t}}]\!]\sigma'[v/\hat{\underline{t}}] \\ & = & \textbf{let } (v,\sigma') = m\sigma_2(t_2)\sigma_2 \textbf{ in } (\sigma'[v/\hat{\underline{t}}](\hat{\underline{t}}),\sigma'[v/\hat{\underline{t}}]) \\ & = & \textbf{let } (v,\sigma') = mn\sigma_2 \textbf{ in } (v,\sigma'[v/\hat{\underline{t}}]) \end{array}$$

where we take the same $\hat{\underline{t}}$ as chosen in *rhs*. Using lemma 1 twice, we have $\mathit{lhs} = \mathit{rhs}$.

The pe-store returned form $\mathcal{PE}[\![MN]\!]\rho\gamma$ in this case is $\gamma_2[\mathsf{apply}(t_1,t_2)/\hat{\underline{t}}]$. It respects $\sigma'[v/\hat{\underline{t}}]$, since γ_2 respects σ_2 and σ' is an extension of σ_2, where $\sigma'(t) = \sigma_2(t)$ for all $t \in \mathit{domain}(\gamma_2)$ from lemma 2.

(Case for $\epsilon : consMN$) Let $\mathcal{E}[\![M]\!]\rho\sigma = (m,\sigma_1)$ and $\mathcal{PE}[\![M]\!]\rho\gamma = (^{\langle\!\langle P_1\rangle\!\rangle}t_1,\gamma_1)$. Assume that γ respects σ. From the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M]\!]\rho\gamma]\!]\sigma = \mathcal{E}[\![M]\!]\rho\sigma$$

we have $\mathcal{E}_{PS}[\![^{\langle\!\langle P_1\rangle\!\rangle}t_1]\!]\sigma = (m,\sigma_1)$ and γ_1 respects σ_1. From lemma 3, we have $\sigma_1(t_1) = m$.

In the same way, let $\mathcal{E}[\![N]\!]\rho\sigma_1 = (n,\sigma_2)$ and $\mathcal{PE}[\![N]\!]\rho\gamma_1 = (^{\langle\!\langle P_2\rangle\!\rangle}t_2,\gamma_2)$. Since γ_1 respects σ_1, from the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![N]\!]\rho\gamma_1]\!]\sigma_1 = \mathcal{E}[\![N]\!]\rho\sigma_1$$

we have $\mathcal{E}_{PS}[\![^{\langle\!\langle P_2\rangle\!\rangle}t_2]\!]\sigma_1 = (n,\sigma_2)$ and γ_2 respects σ_2. From lemma 3, we have $\sigma_2(t_2) = n$.

$$\begin{array}{rcl} \mathit{rhs} & = & \mathcal{E}[\![\epsilon : consMN]\!]\rho\sigma \\ & = & \textbf{let } (v,\sigma') = \mathcal{E}'[\![\epsilon : consMN]\!]\rho\sigma \textbf{ in } (v,\sigma'[v/\hat{\underline{t}}]) \\ & = & \textbf{let } v = \langle\epsilon,\underline{t_m},\underline{t_n}\rangle \textbf{ in } (v,\sigma_2[m/\underline{t_m},n/\underline{t_n},v/\hat{\underline{t}}]) \\ \mathit{lhs} & = & \mathcal{E}'_{PS}[\![\mathcal{PE}[\![\epsilon : consMN]\!]\rho\gamma]\!]\sigma \\ & = & \mathcal{E}_{PS}[\![^{\langle\!\langle P_1,P_2,\underline{t}:\mathsf{pair}(\epsilon,t_1,t_2)\rangle\!\rangle}\hat{\underline{t}}]\!]\sigma \end{array}$$

Since t_1 is fresh and not defined in P_2, we have $\sigma_2(t_1) = \sigma_1(t_1) = m$, and thus

$$\begin{array}{rcl} \mathcal{E}_{PS}[\![^{\langle\!\langle \underline{t}:\mathsf{pair}(\epsilon,t_1,t_2)\rangle\!\rangle}\hat{\underline{t}}]\!]\sigma_2 & = & \textbf{let } (v,\sigma') = \mathcal{E}_S[\![\mathsf{pair}(\epsilon,t_1,t_2)]\!]\sigma_2 \textbf{ in } \mathcal{E}_{PS}[\![^{\langle\!\langle\rangle\!\rangle}\hat{\underline{t}}]\!]\sigma'[v/\hat{\underline{t}}] \\ & = & \textbf{let } (v,\sigma') = (\langle\epsilon,\underline{t_m},\underline{t_n}\rangle,\sigma_2[\sigma_2(t_1)/\underline{t_m},\sigma_2(t_2)/\underline{t_n}]) \textbf{ in } (v,\sigma'[v/\hat{\underline{t}}]) \\ & = & \textbf{let } v = \langle\epsilon,\underline{t_m},\underline{t_n}\rangle \textbf{ in } (v,\sigma_2[\sigma_2(t_1)/\underline{t_m},\sigma_2(t_2)/\underline{t_n},v/\hat{\underline{t}}]) \\ & = & \textbf{let } v = \langle\epsilon,\underline{t_m},\underline{t_n}\rangle \textbf{ in } (v,\sigma_2[m/\underline{t_m},n/\underline{t_n},v/\hat{\underline{t}}]) \end{array}$$

where we take the same $\underline{t_m}$, $\underline{t_n}$, and $\hat{\underline{t}}$ as chosen in *rhs*. Using lemma 1 twice, we have $\mathit{lhs} = \mathit{rhs}$.

The pe-store returned form $\mathcal{PE}[\![\epsilon : consMN]\!]\rho\gamma$ is $\gamma_2[\mathsf{pair}(\epsilon,t_1,t_2)/\hat{\underline{t}}]$. Because $\sigma_2(t_1) = m$ and $\sigma_2(t_2) = n$, and γ_2 respects σ_2, it respects $\sigma_2[m/\underline{t_m},n/\underline{t_n},\langle\epsilon,\underline{t_m},\underline{t_n}\rangle/\hat{\underline{t}}]$.

(Case for $carM$**)** Let $\mathcal{E}[\![M]\!]\rho\sigma = (\langle\epsilon, t_m, t_n\rangle, \sigma_1)$ and $\mathcal{PE}[\![M]\!]\rho\gamma = ({}^{\langle\!\langle P_1\rangle\!\rangle}t_1, \gamma_1)$. Assume that γ respects σ. From the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M]\!]\rho\gamma]\!]\sigma = \mathcal{E}[\![M]\!]\rho\sigma$$

we have $\mathcal{E}_{PS}[\![{}^{\langle\!\langle P_1\rangle\!\rangle}t_1]\!]\sigma = (\langle\epsilon, t_m, t_n\rangle, \sigma_1)$ and γ_1 respects σ_1. From lemma 3, we have $\sigma_1(t_1) = \langle\epsilon, t_m, t_n\rangle$.

$$\begin{aligned} rhs &= \mathcal{E}[\![carM]\!]\rho\sigma \\ &= \textbf{let } (v, \sigma') = \mathcal{E}'[\![carM]\!]\rho\sigma \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\ &= (\sigma_1(t_m), \sigma_1[\sigma_1(t_m)/\underline{\hat{t}}]) \end{aligned}$$

Since γ respects σ and $\sigma_1(t_1) = \langle\epsilon, t_m, t_n\rangle$, if $\gamma_1(t_1)$ is a **pair** Sval, it has the form: $\mathsf{pair}(\epsilon, t_m, t_n)$. In this case, when $\epsilon \in mutable_{car}$,

$$\begin{aligned} lhs &= \mathcal{E}'_{PS}[\![\mathcal{PE}[\![carM]\!]\rho\gamma]\!]\sigma \\ &= \mathcal{E}_{PS}[\![{}^{\langle\!\langle P_1, \underline{\hat{t}}:\mathsf{car}(t_1)\rangle\!\rangle}\underline{\hat{t}}]\!]\sigma \end{aligned}$$

$$\begin{aligned} \mathcal{E}_{PS}[\![{}^{\langle\!\langle \underline{\hat{t}}:\mathsf{car}(t_1)\rangle\!\rangle}\underline{\hat{t}}]\!]\sigma &= \textbf{let } (v, \sigma') = \mathcal{E}_S[\![\mathsf{car}(t_1)]\!]\sigma_1 \textbf{ in } \mathcal{E}_{PS}[\![{}^{\langle\!\langle\rangle\!\rangle}\underline{\hat{t}}]\!]\sigma'[v/\underline{\hat{t}}] \\ &= \textbf{let } (v, \sigma') = (\sigma_1(t_m), \sigma_1) \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\ &= (\sigma_1(t_m), \sigma_1[\sigma_1(t_m)/\underline{\hat{t}}]) \end{aligned}$$

where we take the same $\underline{\hat{t}}$ as chosen in rhs. Using lemma 1, we have $lhs = rhs$.

When $\epsilon \notin mutable_{car}$,

$$\begin{aligned} lhs &= \mathcal{E}'_{PS}[\![\mathcal{PE}[\![carM]\!]\rho\gamma]\!]\sigma \\ &= \mathcal{E}_{PS}[\![{}^{\langle\!\langle P_1, \underline{\hat{t}}:\mathsf{car}(t_1)\rangle\!\rangle}t_m]\!]\sigma \end{aligned}$$

$$\begin{aligned} \mathcal{E}_{PS}[\![{}^{\langle\!\langle \underline{\hat{t}}:\mathsf{car}(t_1)\rangle\!\rangle}t_m]\!]\sigma &= \textbf{let } (v, \sigma') = \mathcal{E}_S[\![\mathsf{car}(t_1)]\!]\sigma_1 \textbf{ in } \mathcal{E}_{PS}[\![{}^{\langle\!\langle\rangle\!\rangle}t_m]\!]\sigma'[v/\underline{\hat{t}}] \\ &= \textbf{let } (v, \sigma') = (\sigma_1(t_m), \sigma_1) \textbf{ in } (\sigma'[v/\underline{\hat{t}}](t_m), \sigma'[v/\underline{\hat{t}}]) \\ &= (\sigma_1(t_m), \sigma_1[\sigma_1(t_m)/\underline{\hat{t}}]) \end{aligned}$$

where we take the same $\underline{\hat{t}}$ as chosen in rhs. Using lemma 1, we have $lhs = rhs$.

If $\gamma_1(t_1)$ is not a **pair** Sval, it means that $\gamma_1(t_1)$ is unknown code. In this case, we can prove in the same way as the above case where $\epsilon \in mutable_{car}$.

The pe-store returned form $\mathcal{PE}[\![carM]\!]\rho\gamma$ is $\gamma_1[\mathsf{car}(t_1)/\underline{\hat{t}}]$. It respects $\sigma_1[\sigma_1(t_m)/\underline{\hat{t}}]$ since γ_1 respects σ_1.

(Case for $setcarMN$**)** Let $\mathcal{E}[\![M]\!]\rho\sigma = (\langle\epsilon, t_m, t_n\rangle, \sigma_1)$ and $\mathcal{PE}[\![M]\!]\rho\gamma = ({}^{\langle\!\langle P_1\rangle\!\rangle}t_1, \gamma_1)$. Assume that γ respects σ. From the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M]\!]\rho\gamma]\!]\sigma = \mathcal{E}[\![M]\!]\rho\sigma$$

we have $\mathcal{E}_{PS}[\![{}^{\langle\!\langle P_1\rangle\!\rangle}t_1]\!]\sigma = (\langle\epsilon, t_m, t_n\rangle, \sigma_1)$ and γ_1 respects σ_1. From lemma 3, we have $\sigma_1(t_1) = \langle\epsilon, t_m, t_n\rangle$.

In the same way, let $\mathcal{E}[\![N]\!]\rho\sigma_1 = (n, \sigma_2)$ and $\mathcal{PE}[\![N]\!]\rho\gamma_1 = ({}^{\langle\langle P_2 \rangle\rangle}t_2, \gamma_2)$. Since γ_1 respects σ_1, from the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![N]\!]\rho\gamma_1]\!]\sigma_1 = \mathcal{E}[\![N]\!]\rho\sigma_1$$

we have $\mathcal{E}_{PS}[\![{}^{\langle\langle P_2 \rangle\rangle}t_2]\!]\sigma_1 = (n, \sigma_2)$ and γ_2 respects σ_2. From lemma 3, we have $\sigma_2(t_2) = n$.

$$\begin{array}{rcl} rhs & = & \mathcal{E}[\![setcar MN]\!]\rho\sigma \\ & = & \textbf{let } (v, \sigma') = \mathcal{E}'[\![setcar MN]\!]\rho\sigma \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\ & = & (n, \sigma_2[n/t_m, n/\underline{\hat{t}}]) \\ lhs & = & \mathcal{E}'_{PS}[\![\mathcal{PE}[\![setcar MN]\!]\rho\gamma]\!]\sigma \\ & = & \mathcal{E}_{PS}[\![{}^{\langle\langle P_1, P_2, \underline{\hat{t}}:\mathsf{setcar}(t_1,t_2)\rangle\rangle}\underline{\hat{t}}]\!]\sigma \end{array}$$

Since t_1 is fresh and not defined in P_2, we have $\sigma_2(t_1) = \sigma_1(t_1) = \langle \epsilon, t_m, t_n \rangle$, and thus

$$\begin{array}{rcl} \mathcal{E}_{PS}[\![{}^{\langle\langle \underline{\hat{t}}:\mathsf{setcar}(t_1,t_2)\rangle\rangle}\underline{\hat{t}}]\!]\sigma_2 & = & \textbf{let } (v, \sigma') = \mathcal{E}_S[\![\mathsf{setcar}(t_1, t_2)]\!]\sigma_2 \textbf{ in } \mathcal{E}_{PS}[\![{}^{\langle\langle\rangle\rangle}\underline{\hat{t}}]\!]\sigma'[v/\underline{\hat{t}}] \\ & = & \textbf{let } (v, \sigma') = (\sigma_2(t_2), \sigma_2[\sigma_2(t_2)/t_m]) \textbf{ in } (v, \sigma'[v/\underline{\hat{t}}]) \\ & = & (\sigma_2(t_2), \sigma_2[\sigma_2(t_2)/t_m, \sigma_2(t_2)/\underline{\hat{t}}]) \\ & = & (n, \sigma_2[n/t_m, n/\underline{\hat{t}}]) \end{array}$$

where we take the same $\underline{\hat{t}}$ as chosen in *rhs*. Using lemma 1 twice, we have $lhs = rhs$.

The pe-store returned form $\mathcal{PE}[\![setcar MN]\!]\rho\gamma$ is $\gamma_2[\mathsf{setcar}(t_1, t_2)/\underline{\hat{t}}]$. It respects $\sigma_2[n/t_m, n/\underline{\hat{t}}]$, since γ_2 respects σ_2.

(Case for $eqMM'NN'$) Let $\mathcal{E}[\![M]\!]\rho\sigma = (m, \sigma_1)$ and $\mathcal{PE}[\![M]\!]\rho\gamma = ({}^{\langle\langle P_1 \rangle\rangle}t_1, \gamma_1)$. Assume that γ respects σ. From the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M]\!]\rho\gamma]\!]\sigma = \mathcal{E}[\![M]\!]\rho\sigma$$

we have $\mathcal{E}_{PS}[\![{}^{\langle\langle P_1 \rangle\rangle}t_1]\!]\sigma = (m, \sigma_1)$ and γ_1 respects σ_1. From lemma 3, we have $\sigma_1(t_1) = m$.

In the same way, let $\mathcal{E}[\![M']\!]\rho\sigma_1 = (m', \sigma_2)$ and $\mathcal{PE}[\![M']\!]\rho\gamma_1 = ({}^{\langle\langle P_2 \rangle\rangle}t_2, \gamma_2)$. Since γ_1 respects σ_1, from the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![M']\!]\rho\gamma_1]\!]\sigma_1 = \mathcal{E}[\![M']\!]\rho\sigma_1$$

we have $\mathcal{E}_{PS}[\![{}^{\langle\langle P_2 \rangle\rangle}t_2]\!]\sigma_1 = (m', \sigma_2)$ and γ_2 respects σ_2. From lemma 3, we have $\sigma_2(t_2) = m'$.

Since t_1 is fresh and not defined in P_2. Suppose that both $\gamma_2(t_1)$ and $\gamma_2(t_2)$ are known and $t_1 = t_2$. Since γ_2 respects σ_2, this means that $m = m' = \langle \epsilon, t'_1, t'_2 \rangle$ or $m = m' = \langle \eta, t'_3, _ \rangle$. In both cases, we use the third induction hypothesis. Let $\mathcal{E}[\![N]\!]\rho\sigma_2 = (n, \sigma_3)$ and $\mathcal{PE}[\![N]\!]\rho\gamma_2 = ({}^{\langle\langle P_3 \rangle\rangle}t_3, \gamma_3)$. Since γ_2 respects σ_2, from the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![N]\!]\rho\gamma_2]\!]\sigma_2 = \mathcal{E}[\![N]\!]\rho\sigma_2$$

we have $\mathcal{E}_{PS}[\![{}^{\langle\langle P_3 \rangle\rangle}t_3]\!]\sigma_2 = (n, \sigma_3)$ and γ_3 respects σ_3. From lemma 3, we have $\sigma_3(t_3) = n$.

$$rhs \quad = \quad \mathcal{E}[\![eqMM'NN']\!]\rho\sigma$$

$$
\begin{array}{rcl}
 & = & \textbf{let } (v,\sigma') = \mathcal{E}'[\![eqMM'NN']\!]\rho\sigma \textbf{ in } (v,\sigma'[v/\hat{\underline{t}}]) \\
 & = & \textbf{let } (v,\sigma') = \mathcal{E}[\![N]\!]\rho\sigma_2 \textbf{ in } (v,\sigma'[v/\hat{\underline{t}}]) \\
 & = & (n,\sigma_3[n/\hat{\underline{t}}]) \\
lhs & = & \mathcal{E}'_{PS}[\![\mathcal{PE}[\![eqMM'NN']\!]\rho\gamma]\!]\sigma \\
 & = & \mathcal{E}_{PS}[\![^{\langle\!\langle P_1,P_2,\underline{\hat{t}}:\mathsf{eq}(t_1,t_2,{}^{\langle\!\langle P_3\rangle\!\rangle}t_3,_)\rangle\!\rangle}t_3]\!]\sigma \\
\mathcal{E}_{PS}[\![^{\langle\!\langle \underline{\hat{t}}:\mathsf{eq}(t_1,t_2,{}^{\langle\!\langle P_3\rangle\!\rangle}t_3,_)\rangle\!\rangle}t_3]\!]\sigma_2 & = & \textbf{let } (v,\sigma') = \mathcal{E}_S[\![\mathsf{eq}(t_1,t_2,{}^{\langle\!\langle P_3\rangle\!\rangle}t_3,_)]\!]\sigma_2 \textbf{ in } \mathcal{E}_{PS}[\![^{\langle\!\langle\rangle\!\rangle}t_3]\!]\sigma'[v/\hat{\underline{t}}] \\
 & = & \textbf{let } (v,\sigma') = \mathcal{E}_{PS}[\![^{\langle\!\langle P_3\rangle\!\rangle}t_3]\!]\sigma_2 \textbf{ in } (\sigma'[v/\hat{\underline{t}}](t_3),\sigma'[v/\hat{\underline{t}}]) \\
 & = & (\sigma_3[n/\hat{\underline{t}}](t_3),\sigma_3[n/\hat{\underline{t}}]) \\
 & = & (n,\sigma_3[n/\hat{\underline{t}}])
\end{array}
$$

where we take the same $\hat{\underline{t}}$ as chosen in *rhs*. Using lemma 1 twice, we have $lhs = rhs$.

The pe-store returned form $\mathcal{PE}[\![eqMM'NN']\!]\rho\gamma$ is $\gamma_3[\mathsf{eq}(t_1,t_2,{}^{\langle\!\langle P_3\rangle\!\rangle}\,t_3,_)/\hat{\underline{t}}]$. It respects $\sigma_3[n/\hat{\underline{t}}]$, since γ_3 respects σ_3.

The case where both $\gamma_2(t_1)$ and $\gamma_2(t_2)$ are known but $t_1 \neq t_2$ can be proved similarly. When one (or both) of $\gamma_2(t_1)$ and $\gamma_2(t_2)$ are unknown, we use the fourth hypothesis as well as the third one presented above. Let $\mathcal{E}[\![N']\!]\rho\sigma_2 = (n',\sigma_4)$ and $\mathcal{PE}[\![N']\!]\rho\gamma_3 = ({}^{\langle\!\langle P_4\rangle\!\rangle}t_4,\gamma_4)$. Because γ_2 respects σ_2 and $\sigma_2(t)$ is undefined for all $t \in domain(\gamma_3)$ and $t \notin domain(\gamma_2)$, γ_3 respects σ_2. From the induction hypothesis:

$$\mathcal{E}'_{PS}[\![\mathcal{PE}[\![N']\!]\rho\gamma_3]\!]\sigma_2 = \mathcal{E}[\![N']\!]\rho\sigma_2$$

we have $\mathcal{E}_{PS}[\![^{\langle\!\langle P_4\rangle\!\rangle}t_4]\!]\sigma_2 = (n',\sigma_4)$ and γ_4 respects σ_4. From lemma 3, we have $\sigma_4(t_4) = n'$.

$$
\begin{array}{rcl}
rhs & = & \mathcal{E}[\![eqMM'NN']\!]\rho\sigma \\
 & = & \textbf{let } (v,\sigma') = \mathcal{E}'[\![eqMM'NN']\!]\rho\sigma \textbf{ in } (v,\sigma'[v/\hat{\underline{t}}]) \\
lhs & = & \mathcal{E}'_{PS}[\![\mathcal{PE}[\![eqMM'NN']\!]\rho\gamma]\!]\sigma \\
 & = & \mathcal{E}_{PS}[\![^{\langle\!\langle P_1,P_2,\underline{\hat{t}}:\mathsf{eq}(t_1,t_2,{}^{\langle\!\langle P_3\rangle\!\rangle}t_3,{}^{\langle\!\langle P_4\rangle\!\rangle}t_4)\rangle\!\rangle}\underline{\hat{t}}]\!]\sigma \\
\mathcal{E}_{PS}[\![^{\langle\!\langle \underline{\hat{t}}:\mathsf{eq}(t_1,t_2,{}^{\langle\!\langle P_3\rangle\!\rangle}t_3,{}^{\langle\!\langle P_4\rangle\!\rangle}t_4)\rangle\!\rangle}\underline{\hat{t}}]\!]\sigma_2 & = & \textbf{let } (v,\sigma') = \mathcal{E}_S[\![\mathsf{eq}(t_1,t_2,{}^{\langle\!\langle P_3\rangle\!\rangle}t_3,{}^{\langle\!\langle P_4\rangle\!\rangle}t_4)]\!]\sigma_2 \\
 & & \textbf{in } \mathcal{E}_{PS}[\![^{\langle\!\langle\rangle\!\rangle}\hat{\underline{t}}]\!]\sigma'[v/\hat{\underline{t}}] \\
 & = & \textbf{let } (v,\sigma') = \mathcal{E}_S[\![\mathsf{eq}(t_1,t_2,{}^{\langle\!\langle P_3\rangle\!\rangle}t_3,{}^{\langle\!\langle P_4\rangle\!\rangle}t_4)]\!]\sigma_2 \\
 & & \textbf{in } (v,\sigma'[v/\hat{\underline{t}}])
\end{array}
$$

Because $\sigma_2(t_1) = m$ and $\sigma_2(t_2) = m'$, if $m = m'$, then

$$
\begin{array}{rcl}
rhs & = & \textbf{let } (v,\sigma') = \mathcal{E}'[\![eqMM'NN']\!]\rho\sigma \textbf{ in } (v,\sigma'[v/\hat{\underline{t}}]) \\
 & = & (n,\sigma_3[n/\hat{\underline{t}}]) \\
lhs & = & \textbf{let } (v,\sigma') = \mathcal{E}_S[\![\mathsf{eq}(t_1,t_2,{}^{\langle\!\langle P_3\rangle\!\rangle}t_3,{}^{\langle\!\langle P_4\rangle\!\rangle}t_4)]\!]\sigma_2 \\
 & & \textbf{in } (v,\sigma'[v/\hat{\underline{t}}]) \\
 & = & (n,\sigma_3[n/\hat{\underline{t}}])
\end{array}
$$

where we take the same $\hat{\underline{t}}$ as chosen in *rhs*. Using lemma 1 twice, we have $lhs = rhs$.

The pe-store returned form $\mathcal{PE}[\![eqMM'NN']\!]\rho\gamma$ in this case is $\gamma_4[\mathsf{eq}(t_1,t_2,{}^{\langle\!\langle P_3\rangle\!\rangle}\,t_3,{}^{\langle\!\langle P_4\rangle\!\rangle}\,t_4)/\hat{\underline{t}}]$. It respects $\sigma_3[n/\hat{\underline{t}}]$, since γ_3 respects σ_3 and $\sigma_3(t)$ is undefined for all $t \in domain(\gamma_4)$ and $t \notin domain(\gamma_3)$.

The case for $m \neq m'$ can be proved similarly. $\square$