TECHNICAL REPORT 94-11

Roles of a Partial Evaluator for the Reflective Language Black

Kenichi Asai Satoshi Matsuoka Akinori Yonezawa

May 25, 1994

Department of Information Science
Faculty of Science, University of Tokyo

7–3–1 Hongo, Bunkyo-Ku Tokyo, 113 Japan

TECHNICAL REPORT 94-11

TITLE
Roles of a Partial Evaluator for the Reflective Language Black

AUTHORS
Kenichi Asai, Satoshi Matsuoka, and Akinori Yonezawa

KEY WORDS AND PHRASES
reflection, partial evaluation, metacircular interpreter, direct execution, interpretation, infinite regression

ABSTRACT
A reflective language allows us to access its interpreter to dynamically modify or extend the language semantics from within the same language. In the previous paper, we implemented a reflective language called *Black*, and showed that a directly executable interpreter can be implemented by *duplicating* and *partially evaluating* the metalevel interpreters without losing changeability. In this paper, we will concentrate on the partial evaluator and show that it can be used not only to construct the default reflective interpreter but also as a compiler of reflective languages. Since the language semantics may change in reflective languages, we cannot use a specific compiler; instead, we use the partial evaluator, which can take the semantics of the language into account.

REPORT DATE	**WRITTEN LANGUAGE**
May 25, 1994	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
10	9

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT

DISTRIBUTION STATEMENT
First issue 35 copies.

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku Tokyo, 113 Japan

Roles of a Partial Evaluator for the Reflective Language Black

Kenichi Asai Satoshi Matsuoka* Akinori Yonezawa

Abstract

A reflective language allows us to access its interpreter to dynamically modify or extend the language semantics from within the same language. In the previous paper, we implemented a reflective language called *Black*, and showed that a directly executable interpreter can be implemented by *duplicating* and *partially evaluating* the metalevel interpreters without losing changeability. In this paper, we will concentrate on the partial evaluator and show that it can be used not only to construct the default reflective interpreter but also as a compiler of reflective languages. Since the language semantics may change in reflective languages, we cannot use a specific compiler; instead, we use the partial evaluator, which can take the semantics of the language into account.

1 Introduction

A reflective language allows us to access its interpreter to dynamically modify or extend the language semantics from within the same language. When we are programming (especially debugging), we sometimes want to slightly modify the language semantics. For example, suppose we are test-executing an incomplete Lisp program. If the Lisp interpreter encounters an unbound function which we have not yet implemented, it will report an error. In such a case, however, we want the Lisp interpreter to ask for the return value and proceed execution to see if the rest of the program goes well, rather than report an error and stop. Reflective languages allow us such modifications in a simple and clean way.

In reflective languages, user programs running at baselevel (level 0) are interpreted by an interpreter running at metalevel (level 1). The interpreter at level 1 is in turn interpreted by an interpreter running at level 2, and so on. Conceptually, there is an infinite tower of interpreters. Reflective languages permit us to go up and down freely in the tower, and modify or extend the language semantics.

However, it is impossible to implement this infinite tower structure directly. In conventional Lisp-based reflective languages, such as Brown[9] and Blond[3], the tower is simulated by creating *directly executable* interpreters *lazily*, instead of interpreting them

*Department of Mathematical Engineering, Faculty of Engineering, The University of Tokyo

explicitly. Although this method achieves high efficiency, it becomes impossible to modify the interpreters, because they are directly executed in machine code.

In [1], we implemented a reflective language called *Black*, and showed that a directly executable interpreter can be implemented by *duplicating* and *partially evaluating* the metalevel interpreters without losing changeability. In this method, a partial evaluator plays an important role to produce an efficient interpreter. In this paper, we will describe the roles of the partial evaluator and show that it can be used not only to construct the default reflective interpreter but also as a compiler of reflective languages. Since the language semantics may change in reflective languages, we cannot use a specific compiler; instead, we use the partial evaluator, which can take the semantics of the language into account.

2 The reflective language Black

In this section, we briefly present the language Black, and discuss the difference between interpretation and direct execution.

2.1 The infinite tower

The syntax of Black is the same as Scheme with one additional reflective construct `(exec-at-metalevel E)`. As in Brown and Blond, there is an infinite tower of interpreters in Black. The construct `(exec-at-metalevel E)` is used to evaluate the expression `E` one level above.

Figure 1 shows the level n interpreter which interprets Black programs of level $n-1$[1]. The interpreter is basically a usual continuation passing style (CPS) Scheme interpreter, except for the function `eval-EatM`$_n$[2]. `eval-EatM`$_n$ is defined to interpret the `exec-at-metalevel` construct. By calling `eval`$_{n+1}$, it evaluates the body `E` of `(exec-at-metalevel E)` at level n, that is, at the same level as where the `eval-EatM`$_n$ runs. Let us write this interpreter as $\mathtt{I}_n(\cdot)$. Then, a Black program exp is conceptually interpreted by an infinite tower of $\mathtt{I}_n(\cdot)$ as follows (omitting continuations for clarity):

$$\ldots \mathtt{I}_3(\mathtt{I}_2(\mathtt{I}_1(exp, env_1), env_2), env_3) \ldots$$

This is the overview of the Black interpreter, which can be observed from user programs. Each $\mathtt{I}_n$ defines the operational semantics of the language L_{n-1} of level $n-1$ and interprets $\mathtt{I}_{n-1}$ written in L_{n-1}. At the bottom of the tower, $\mathtt{I}_1$ interprets user programs. Note that $\mathtt{I}_n$ is not always interpreted by $\mathtt{I}_{n+1}$, since in `eval-EatM`$_n$, $\mathtt{I}_n$ contains a free variable env_{n+1} and a call to `eval`$_{n+1}$ ($\mathtt{I}_{n+1}$), which are not defined in $\mathtt{I}_n$. Instead of being interpreted by $\mathtt{I}_{n+1}$, this part directly calls $\mathtt{I}_{n+1}$ to execute the body of `exec-at-metalevel` expressions one level above. So to speak, `eval-EatM`$_n$ has a "hole" to upper levels, which cannot be understood from the ordinary tower structure.

[1] Subscripts attached to function names indicate the level of the function.

[2] `EatM` is an abbreviation of `exec-at-metalevel`.

```
(define (eval_n exp_n env_n cont_n)
  (cond ((or (number? exp_n) (boolean? exp_n)) (cont_n exp_n))
        ((symbol? exp_n)                       (eval-var_n exp_n env_n cont_n))
        ((eq? (car exp_n) 'quote)              (eval-quote_n exp_n env_n cont_n))
        ((eq? (car exp_n) 'if)                 (eval-if_n exp_n env_n cont_n))
        ((eq? (car exp_n) 'set!)               (eval-set!_n exp_n env_n cont_n))
        ((eq? (car exp_n) 'lambda)             (eval-lambda_n exp_n env_n cont_n))
        ((eq? (car exp_n) 'exec-at-metalevel) (eval-EatM_n exp_n env_n cont_n))
        (else (eval-list_n exp_n env_n
                           (lambda (l) (apply_n (car l) (cdr l) cont_n))))))
(define (eval-if_n exp_n env_n cont_n)
  (eval_n (car (cdr exp_n)) env_n
          (lambda (pred)
            (if pred (eval_n (car (cdr (cdr exp_n))) env_n cont_n)
                     (eval_n (car (cdr (cdr (cdr exp_n)))) env_n cont_n)))))
(define (eval-lambda_n exp_n env_n cont_n)
  (let ((lambda-body (cdr (cdr exp_n)))
        (lambda-params (car (cdr exp_n))))
     (cont_n (lambda args
               (eval-list_n lambda-body
                            (extend env_n lambda-params args)
                            (lambda (l) (car (last-pair l))))))))
(define (eval-EatM_n exp_n env_n cont_n)
  (eval_{n+1} (car (cdr exp_n)) env_{n+1} cont_n))
(define (apply_n operator operand cont_n)
  (if (procedure? operator)
      (cont_n (scheme-apply operator operand))
      (error "Not a function" operator)))
```

Figure 1: The overview of level n interpreter $\mathtt{I}_n$

2.2 Implementation difficulties

The infinite tower of interpreters presented above is unimplementable, since all interpreters need to be interpreted by the interpreter one level above, causing infinite regression. Jefferson and Friedman implemented in [5] a finite tower of interpreters by simply limiting the number of levels we can go up. Although we need only several levels in practice, this does not help us to understand how reflective languages with an infinite tower work and can be implemented. Another problem of their approach lies in efficiency: because all the interpreters except for the top one are actually interpreted by the interpreter one level above, it is extremely slow.

Brown[9] and Blond[3] achieved the infinite tower of interpreters efficiently by creating *directly executed* interpreters *lazily*. This method, however, does not allow us to redefine the interpreters themselves, since they are directly executed in the machine language. We can

freely go up and down in the tower of interpreters and modify environments, continuations, etc., but cannot alter the interpreters themselves. Although the direct execution method is essential for the efficient execution of reflective languages, it cannot be directly applied to Black, where the interpreters may be changed. To implement reflective languages like Black, we need to clarify the distinction between the interpreted code and the directly executed code, and use both to achieve changeability and high efficiency.

2.3 Interpretation and direct execution

Consider a user program `P` written in Scheme. If `P` is interpreted by a Scheme interpreter `I` as in `I(P,env)`, we can change any part of `P`. For example, if we want to redefine a function `f` in `P` to `g`, all we have to do is `(set! f g)`. After that, the value of `f` in `env` is changed to `g`. Because the environment is looked up for every function call in `P`, it is this `g` that is executed when `f` is called.

We can make more sophisticated changes using reflective facilities. Imagine that we want to alter the evaluation order of arguments. It is typically written within `eval-list` (or *evlis*) in `I`. So, if we replace `eval-list` with a new user-defined `my-eval-list` using reflective facilities[3], the evaluation order of arguments changes to the one defined in `my-eval-list` afterwards. Because `P` is actually interpreted by `I`, we can arbitrarily change the behavior of `P`, if we are allowed to modify `I`. This can be naturally understood, since the interpreter `I` has all the information of `P` for interpretation.

What if we compile `P` into a machine code $\mathcal{P}$[4] and directly execute it? In this case, all the changes mentioned above become impossible, in general, because $\mathcal{P}$ is directly executed. Even if we stop the execution of $\mathcal{P}$ and execute `(set! f g)`, `f` would remain the same, since $\mathcal{P}$ is compiled and does not look up the environment[5]. Redefinition of `eval-list` does not affect the behavior of $\mathcal{P}$ neither, because $\mathcal{P}$ is no longer interpreted by `I`. Direct execution achieves high efficiency by throwing away the possibility that programs may be changed. If we want to allow all the changes mentioned above, we need to interpret `P` explicitly using `I`. In other words, to construct an efficient interpreter that allows changes, we have to restrict the changes we can make.

In Black, we set the restriction that only changes on the metalevel environment will affect the behavior of the baselevel compiled code. That is, if we change the metalevel environment by using `set!`, the behavior of compiled code at baselevel will be changed. However, even if we redefine functions of the metalevel interpreter (such as `eval-list`), the baselevel compiled code will behave the same as before. Specifically, this means that a compiled code can correctly handle a program which redefines some of its functions during its execution, but cannot handle a program which modifies its metalevel interpreter. We

[3] In Black, this can be written as: `(exec-at-metalevel (set! eval-list my-eval-list))`

[4] In this paper, we consistently use the typewriter font (such as `P`, `I`, etc.) for interpreted programs, and the calligraphic font ($\mathcal{P}$, $\mathcal{I}$, etc.) for directly executable compiled programs.

[5] This is exactly the case in SML, where we have to recompile the whole program after we redefine some of its functions. On the other hand, we need no recompilation in MIT-Scheme compiler. However, the compiled code always has to look up an environment. Thus, a function call can not be compiled into a single `jmp` instruction.

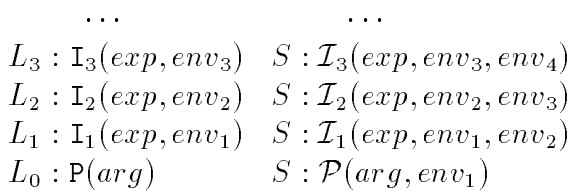

Figure 2: Overview of Black

think that this restriction is reasonable, because we can replace any compiled function with a user-defined one by changing the environment using `set!`. So, we can change the compiled metalevel interpreter to modify the behavior of baselevel interpreted programs. We cannot modify the behavior of baselevel compiled programs, but this is natural since they are *compiled*. Furthermore, we can obtain the compiled programs which reflect the modified semantics by using a compiler we will present in Section 3.3. With this restriction, we can compile a program into an efficient code without losing the ability to change the language semantics.

3 Compiling without losing changeability

In this section, we describe how the Black interpreter is directly executed without losing changeability. Based on the idea shown in [1], we will further show that the partial evaluator used to obtain the Black interpreter can be used as a compiler for reflective languages. The use of a partial evaluator rather than a specific compiler is necessary, as the semantics of the language may change in reflective languages.

3.1 The structure of Black interpreter

Before constructing the Black interpreter, we show the overview of the Black interpreter we are going to make. See Figure 2. The left column shows the overview observed from the user. $L_n : \mathtt{I}_n(exp, env_n)$ indicates that the interpreter $\mathtt{I}_n$ is written in the language L_n. It defines the language L_{n-1} and interprets $\mathtt{I}_{n-1}$. Initially, all the L_n's are the same language, but may be changed if we modify or extend the language semantics using `exec-at-metalevel`. The right column in the Figure shows the directly implemented interpreter which are actually used to interpret Black programs. $S : \mathcal{I}_n(exp, env_n, env_{n+1})$ indicates that the interpreter $\mathcal{I}_n$ is written in the language S. S is the language used to implement directly executable interpreters. In the case of Black, it is the ordinary non-reflective Scheme.

A non-compiled baselevel user program `P` is usually interpreted by $\mathcal{I}_1$. If `(exec-at-metalevel E)` is executed in `P`, $\mathcal{I}_1$ calls $\mathcal{I}_2$ to interpret the body `E` at the metalevel. Suppose that a `set!` statement in `E` redefines a function of $\mathcal{I}_1$. In $\mathcal{I}_2$, this causes a change in env_2. Since env_2 is shared by $\mathcal{I}_1$, $\mathcal{I}_1$ can 'detect' the change, and it uses the redefined function

afterwards, i.e., $\mathcal{I}_1$ is modified. The inter-level sharing of environments is important here. Through the sharing, changes made on the environment of $\mathcal{I}_2$ will affect the behavior of $\mathcal{I}_1$. After $\mathcal{I}_1$ is modified, the user program `P` is interpreted by the new modified $\mathcal{I}_1$.

If we compile `P` into $\mathcal{P}$ using the compiler which will be described in Section 3.3, $\mathcal{P}$ is directly executed instead of being interpreted by $\mathcal{I}_1$. In this case, changes made on env_1 will affect the behavior of $\mathcal{P}$ in Black, whereas redefinition of functions in $\mathcal{I}_1$ will not. That is, if a function in $\mathcal{P}$ is redefined using `set!`, the new function will be used afterwards, since $\mathcal{P}$ shares env_1 with $\mathcal{I}_1$. However, redefinition of a function in $\mathcal{I}_1$ using `exec-at-metalevel` and `set!` will not affect the behavior of $\mathcal{P}$, as $\mathcal{P}$ is not interpreted by $\mathcal{I}_1$. To make the change valid, we need to recompile `P` using the new $\mathcal{I}_1$.

3.2 Constructing the default $\mathcal{I}_n$

We now construct the default $\mathcal{I}_n$. The goal of this section is to obtain $\mathcal{I}_n$, which runs almost as efficiently as the ordinary metacircular interpreter $\mathtt{I}_n$, and still allows us to redefine functions in it. We will construct $\mathcal{I}_1$ as an example, but other $\mathcal{I}_n$ can be obtained in the same way by just changing subscripts.

The final output $\mathcal{I}_1$ turns out to be very much similar to $\mathtt{I}_1$ except that it contains accesses to env_2, which would have been performed if $\mathtt{I}_1$ were actually interpreted by $\mathtt{I}_2$. In other words, $\mathcal{I}_1$ is the result of specializing $\mathtt{I}_2$ on $\mathtt{I}_1$, residualizing all the accesses to env_2:

$$\mathcal{I}_1(exp, env_1, env_2) = PE(\mathtt{I}_2(\mathtt{I}_1(exp, env_1), env_2))$$

Here, PE is a partial evaluator. We partially evaluate $\mathtt{I}_2$ on $\mathtt{I}_1$ with unknown parameters exp, env_1, and env_2[6]. Generally, the result of this partial evaluation is $\mathtt{I}_1$ itself, since $\mathtt{I}_2$ is a meta-circular interpreter. To obtain $\mathcal{I}_1$, we add a condition that env_2 should be always treated as unknown. Even though env_2 is initially unknown, the partial evaluator can deduce its structure as pairs of a function name and its body[7], and use the information to generate more specialized code, because the series of function definitions of $\mathtt{I}_1$ is known to the partial evaluator. By forbidding the use of this information, we can force all the accesses to env_2 to residualize, and obtain an directly executable interpreter which allows us the redefinition of functions in it. Using a semi-automatic online partial evaluator, we have actually obtained $\mathcal{I}_n$ to construct the Black interpreter.

Figure 3 shows $\mathtt{eval\text{-}if}_n$ of the obtained $\mathcal{I}_n$. It has an extra argument $\mathtt{env}_{n+1}$. The structure of the function is exactly the same as that of $\mathtt{eval\text{-}if}_n$ in Figure 2, but it contains many accesses to $\mathtt{env}_{n+1}$. This is because functions, such as $\mathtt{eval}_n$, may be changed by the user. By looking up $\mathtt{env}_{n+1}$ each time, changes made on $\mathtt{env}_{n+1}$ can always be captured. The code in Figure 3 may be inefficient, since it looks up $\mathtt{env}_{n+1}$ even when calling primitive functions, which are hardly ever redefined in practice. Section 3.4 shows how to remove such unnecessary indirections.

[6] The function call to $\mathtt{eval}_2$ in $\mathtt{I}_1$ is also treated as an unknown function. It is manually connected to $\mathcal{I}_2$ when the tower of interpreters is constructed.

[7] Assuming that an environment is represented as an association list.

```
(define (eval-if_n exp_n env_n cont_n env_{n+1})
  ((get 'eval_n env_{n+1})
      ((get 'car env_{n+1}) ((get 'cdr env_{n+1}) exp_n))
      env_n
      (lambda (pred)
        (if pred ((get 'eval_n env_{n+1}) ((get 'car env_{n+1})
                                           ((get 'cdr env_{n+1})
                                              ((get 'cdr env_{n+1}) exp_n)))
                                          env_n cont_n)
                 ((get 'eval_n env_{n+1}) ((get 'car env_{n+1})
                                           ((get 'cdr env_{n+1})
                                              ((get 'cdr env_{n+1})
                                                 ((get 'cdr env_{n+1}) exp_n))))
                                          env_n cont_n)))))
```

Figure 3: `eval-if`$_n$ of $\mathcal{I}_n$

Notice that we used a 'trick' here: we have specialized $\mathtt{I}_2$ instead of $\mathcal{I}_2$ to obtain $\mathcal{I}_1$. Since $\mathtt{I}_2$ is written in L_2, the specialized program also becomes an L_2-program, not an S-program. This means that the obtained program may contain `exec-at-metalevel`, and thus cannot be directly executed as an S-program. This does not happen because $\mathtt{I}_1$ is written without using `exec-at-metalevel`, which means that `eval-EatM`$_2$ in $\mathtt{I}_2$ will never be called. Without `eval-EatM`$_2$, $\mathtt{I}_2$ is an ordinary directly executable Scheme program, and so is $\mathcal{I}_1$.

3.3 Compiling using a partial evaluator

In the previous section, we have constructed the directly executable and modifiable interpreter $\mathcal{I}_n$ using a partial evaluator. In this section, we use the partial evaluator as a compiler to compile both user programs and modified interpreters.

Although the Black interpreter we have made in the previous section runs efficiently at first, it becomes slower as (parts of) the interpreter is redefined. For example, if we entirely replace $\mathcal{I}_1$ with a user-defined $\mathtt{I}'_1$, the user program `P` will now be interpreted by $\mathtt{I}'_1$ which is in turn interpreted by $\mathcal{I}_2$, which is quite inefficient. To avoid this interpretation overhead, we compile $\mathtt{I}'_1$ into $\mathcal{I}'_1$ by specializing $\mathcal{I}_2$ on $\mathtt{I}'_1$[8]:

$$\mathcal{I}'_1(exp, env_1, env_2) = PE(\mathcal{I}_2(\mathtt{I}'_1(exp, env_1), env_2, env_3))$$

Here, exp, env_1, and env_2 are unknown parameters, while env_3 is a known parameter which holds the body of level 2 functions when the compilation takes place[9]. We use the

[8]This time, we use $\mathcal{I}_2$ instead of $\mathtt{I}_2$, since $\mathcal{I}_2$ has been already obtained. This avoids the problem of regarding $\mathtt{I}_2$ as an S-program.

[9]The call to $\mathcal{I}_3$ in $\mathcal{I}_2$ should also be residualized as in the previous case.

value of env_3 as a known value, because we could hardly partially evaluate $\mathcal{I}_2$ without it: Let us see Figure 3 again. All the function parts of functional application have the form `((get '<function-name> env`$_3$`) <args>)`. Without the value of env_3, we cannot further partially evaluate this expression, since the function part is unknown. This results in a very poor residualized program, which is almost the same as $\mathcal{I}_2(\mathtt{I'}_1(exp, env_1), env_2, env_3)$ itself. To get an efficient $\mathcal{I}'_1$, we need the value of env_3.

Because we have used the value of env_3 as a known value, changes made on env_3 after the partial evaluation will no longer affect the behavior of $\mathcal{I}'_1$. This actually corresponds to the restriction we have made before: In Section 2.3, we limited possible changes on compiled programs to the ones on the metalevel environment env_2 only. Compiled programs will ignore changes on the metalevel interpreter itself, or in other words, the metametaenvironment env_3. By using the value of env_3 at partial evaluation time and residualizing only env_2, we can obtain the efficient interpreter $\mathcal{I}'_1$ that ignores the changes on env_3. Without the value of env_3, we cannot obtain the efficient $\mathcal{I}'_1$.

The partial evaluator can be used to compile user programs, too. To compile a user program `P`, we specialize $\mathcal{I}_1$ on `P` as follows:

$$\mathcal{P}(arg, env_1) = PE(\mathcal{I}_1(\mathtt{P}(arg), env_1, env_2))$$

where exp and env_1 are unknown parameters, while env_2 holds the value of env_2 at partial evaluation time. Note that $\mathcal{I}_1$ need not be always the default $\mathcal{I}_1$. $\mathcal{I}_1$ may be the modified interpreter $\mathcal{I}'_1$. To compile programs of reflective languages where the language semantics may change, we need to use a partial evaluator, so that we can reflect the current language semantics in the compiled code.

3.4 Removing unnecessary indirection

The compiled programs obtained in the previous two sections are sometimes inefficient because all the accesses to the metaenvironment are residualized. In Figure 3, for example, env_{n+1} is accessed to call even primitive functions, such as `car` or `cdr`. Since primitive functions are usually not modified, we want `eval-if`$_n$ to directly call the primitive functions without looking up env_{n+1}.

To control the access to the metaenvironment, we weaken the restriction that the metaenvironment should always be unknown. Instead, the pairs that contain functions which will never be modified are treated to be known. For example, to make primitive function calls faster, the pairs for primitive functions in the metaenvironment are used in partial evaluation process, while other pairs remain unknown to be residualized. In the partial evaluator we have developed, the metaenvironment is bound to something like the following structure: `((car . #<car>) (cdr . #<cdr>) ... (write . #<write>) . unknown-env`$_{n+1}$`)`, where `unknown-env`$_{n+1}$ indicates that the rest of the environment is unknown. Using this environment, we can get the optimized result shown in Figure 4. By controlling the known parts of metaenvironments, we can avoid unnecessary indirection.

```
(define (eval-if_n exp_n env_n cont_n env_{n+1})
  ((get 'eval_n env_{n+1})
      (car (cdr exp_n))
      env_n
      (lambda (pred)
        (if pred ((get 'eval_n env_{n+1}) (car (cdr (cdr exp_n)))
                                          env_n cont_n)
                 ((get 'eval_n env_{n+1}) (car (cdr (cdr (cdr exp_n))))
                                          env_n cont_n)))))
```

Figure 4: `eval-if`$_n$ of $\mathcal{I}_n$ after removing unnecessary indirection

4 Related work

There seems to be only a few papers which treats both reflection and partial evaluation. Danvy pointed out in [2] the similarity between a reflective tower and the Futamura Projection[4]. This is in contrast to our approach, in which we took a partial evaluator as a tool to implement reflective languages.

The use of partial evaluation in a reflective system is mentioned by Ruf[8]. He proposes to use partial evaluation to improve the performance of programs written under 'open' language implementations. In this paper, by restricting and clarifying the changeable parts of 'open' implementations, we applied partial evaluation to get an efficient modifiable interpreter.

In concurrent object-oriented community, Masuhara et al.[7] achieved high efficiency in the language ABCL/R2 by partially compiling scripts. The approach taken here is very similar to ours, but done in a rather ad-hoc manner. We hope our work would give us a clear explanation of their approach.

In the previous paper[1], we showed that the Black interpreter can be implemented by *duplicating* and *partially evaluating* the metalevel interpreters. In this paper, we concentrated on the partial evaluator and showed that it can be used as a compiler in reflective languages.

Recently, research on a compiler in reflective languages is done by Kiczales et al.[6]. Using Meta Object Protocol (MOP) based approach, they allow users to participate in a compilation process. Since our system currently does not show the partial evaluator to users, the compilation process can not be modified. However, we believe that it is necessary to make it open to produce a highly efficient compiled code.

5 Conclusion

In this paper, we have shown that a partial evaluator can be used for both constructing reflective interpreters and compiling them. Since the language semantics may change in reflective languages, we used the partial evaluator to take the semantics of the language

into account, instead of using a specific compiler. We are now directing our research into two ways. First, we are designing a general-purpose partial evaluator to be used as a fully automatic compiler in our reflective language Black. Secondly, we are developing a mechanism to display the partial evaluator to users, so that it can be modified to produce more efficient compiled code.

Acknowledgements

We would like to thank Hidehiko Masuhara for his suggestions and discussions on reflection. Thanks also go to Erik Ruf, whose comments on partial evaluation improved this paper in various ways.

References

[1] Asai, K., S. Matsuoka, and A. Yonezawa "Duplication and Partial Evaluation — A General Implementation Scheme for Reflective Languages," Submitted for publication (October 1993).

[2] Danvy, O. "Across the bridge between reflection and partial evaluation," In D. Djørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, pp. 83–116, North-Holland (1988).

[3] 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).

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

[5] 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).

[6] Kiczales, G., J. Lamping, and A. Mendhekar "What A Metaobject Protocol Based Compiler Can Do For Lisp," Xerox PARC technical report (January 1994).

[7] Masuhara, M., S. Matsuoka, T. Watanabe, and A. Yonezawa "Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently," *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*, pp. 127–145, Vancouver, (October 1992).

[8] Ruf E. "Partial Evaluation in Reflective System Implementations," In *Proceedings of the OOPSLA'93 Workshop on Reflection and Metalevel Architecture*, Washington D.C., (October 1993).

[9] 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).