Dynamic Compilation of a Reflective Language Using Run-Time Specialization

Hidehiko Masuhara
Graduate School of
Arts and Sciences
University of Tokyo
masuhara@acm.org

Yuuya Sugita
Graduate School of
Arts and Sciences
University of Tokyo

Akinori Yonezawa
Department of
Information Science
University of Tokyo
yonezawa@is.s.u-tokyo.ac.jp

Abstract

In reflective languages, application programs can customize a language system that executes the application programs. Our premise is that this customizability of reflective languages can be a basic mechanisms of software evolution. In this paper, we present a simple architecture of a reflective language that can dynamically *select meta-interpreters, and a* dynamic compilation *scheme by using* run-time specialization *(RTS) techniques, which could also be useful to dynamically optimize systems with mechanisms of dynamic software evolution. Our prototype system showed that dynamically compiled reflective programs run more than four times faster than the interpreted ones, and that compilation processes are fast enough to be invoked at run-time. Compared to statically compiled programs, however, dynamically compiled ones yet have 20–30% overheads. We also discuss this problem.*

1. Introduction

In reflective languages, the semantics of a language (*i.e.*, how an application program is executed by the language system) can be customized from the application program[14, 21, 23]. Those languages expose their implementations as *meta-circular interpreters,* which can be considered as a basic mechanism of software evolution[1, 17]. We therefore believe that studies on design and implementation of reflective languages are also beneficial to systems with mechanisms of software evolution.

Early reflective languages allow *dynamic* customizations—meta-interpreters can be modified from running application programs. In other words, these languages have mechanisms of *dynamic software evolution.* Although most studies on software evolution are aiming at static evolution, studies on dynamic evolution should also be important for modern software systems that have dynamic nature, such as mobile agents, software components, and global computing.

Many practical reflective languages, however, omit the ability of dynamic customization. This is because the ability usually poses tremendous amount of overheads. In other words, existing implementation techniques of reflective languages, such as *compilation by using partial evaluation*[15, 16] and *compile-time reflection*[3, 8], only support *static* customizations.

In this paper, we propose an approach to support dynamic customization in reflective languages by *dynamically* compiling reflective programs using *run-time specialization* (RTS) techniques. The primary goal of the paper is to present a basic design of the language system, and to reveal problems of the RTS techniques by measuring the performance of the system. As we will see in Section 5, compiled programs in our prototype system have a certain amount of overheads. We also discuss several approaches to reduce them.

The rest of the paper is organized as follows. Section 2 introduces reflective languages and their compilation techniques by using partial evaluation. Section 3 introduces run-time specialization techniques. Section 4 presents the design of our reflective language and its implementation scheme by using run-time specialization techniques. Section 5 shows performance of our prototype implementation. Section 6 discusses performance problems in the compiled programs and several approaches to improve the performance. Section 7 concludes the paper.

0-7695-0906-1/01 $10.00 © 2001 IEEE

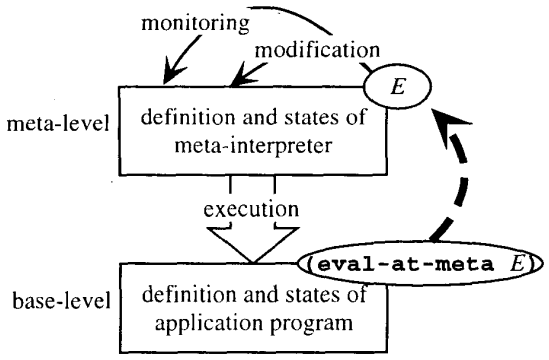

Figure 1. Accessing a Meta-circular Interpreter in a Reflective Language

2. Reflective Languages and Compilation Techniques

2.1. Reflective Languages

In reflective languages, application programs can customize an implementation of the language. The customization is achieved by modifying or extending a *meta-circular interpreter*, which is an abstracted implementation of the language.

In order to access a meta-circular interpreter from application programs, reflective languages usually provide several special operators. In reflective language Black[2], for example, when an expression `(eval-at-meta` E`)` is evaluated in an application program, E is actually evaluated at the *meta-level*, where the meta-circular interpreter is running (Figure 1). The expression E can monitor internal states of the system by calling predefined functions or reading global variables defined at the meta-level. It can also modify the definition of the meta-circular interpreter by replacing (a part of) the definition of the interpreter.

Thus far, reflection have been studied in a number of programming languages including functional[21, 22], object-oriented[3, 11, 14], concurrent object-oriented[8, 20, 23], and logic programming languages.

2.2. Implementation Techniques of Reflective Languages: Compiling by Using Partial Evaluation

2.2.1 Interpretive Execution

One of the major challenges in reflective languages is efficient execution because those languages fundamentally rely on *interpreters* to extend or modify the language. Because the semantics of the language can be changed by customizing meta-interpreters, it is difficult to compile application programs. Early reflective languages are implemented by actually running interpreters. As a result, programs run tremendously (by the orders of magnitude) slower than the compiled programs in non-reflective languages.

One of the approaches to reduce the overheads is to compile reflective programs by using *partial evaluation*. Below, we first introduce partial evaluation, and then shows how reflective programs are compiled by using partial evaluation.

2.2.2 Partial Evaluation

Partial evaluation is a technique that automatically specializes a program with respect to some of inputs to the program[6, 9]. Let $p(x, y)$ be a program that takes two parameters x and y. Partial evaluation of p takes a value (*e.g.*, v) to part of its parameters (*e.g.*, x), performs all the computation that depends on the value of x, and leaves a specialized program that contains computations depending on values other than x. We will denote the process as:

$$S(p, v) = p_v, \tag{1}$$

where S and p_v are the partial evaluator and the specialized program. The specialized program, which takes the rest of the parameters, performs remaining computation and returns a result that would have been returned by the original program:

$$p(v, w) = p_v(w). \tag{2}$$

By removing computations that depend on x in p, the execution of $p_v(w)$ is usually faster than that of $p(v, w)$.

For example, let $p(x, y)$ be a function that computes x'th power of y. Assume we found that the parameter x is usually 3 in a program. In this case, we can specialize p with respect to x with 3 by using a partial evaluator, and obtain and a function $p_3(y)$ that computes the third power of y. In this function, there are no computation depending on x; p_3 is thus faster than p.

2.2.3 The First Futamura Projection

Futamura showed that partial evaluation of an interpreter can compile programs written in the language whose semantics is defined by the interpreter. This compilation process is called the *first Futamura projection*[6].

Let i^L be an interpreter of language I written in language L, and p^I be a program written in I. An interpretive execution of the program is to apply p and p's input x to i, which is written as $i^L(p^I, x)$.

Partial evaluation of i with respect to p generates specialized version of i:

$$S(i^L, p^I) = i_p^L \quad . \tag{3}$$

Since i_p^L is a specialized version of i, i_p^L is a program written in L. This means that a program written in I is compiled into L. By using a compiler $C^{L\to M}$ (where the superscript denotes that it compiles from a language L into a machine language M), the interpreted program can be eventually compiled into a machine language:

$$C^{L\to M}(S(i^L, p^I)) = i_p^M \quad . \tag{4}$$

2.2.4 Compilation in Reflective Languages

There are several reflective language systems that successfully compile programs by applying the first Futamura projection[2, 15, 16]. One of the most crucial premises in their compilation scheme is that customization of a meta-interpreter and execution of a body of an application program are *interleaved*. This is because they compile the body part by partially evaluating the customized interpreter with respect to the body part.

In order to make the above premise true, those reflective language systems restrict the ability of dynamic customization. For example, ABCL/R3[15, 16], which can compile programs by using partial evaluation, requires that both base- and meta-level programs are known before compilation.

3. Run-time Specialization Techniques

As mentioned above, compilation using partial evaluation makes reflective languages difficult to support dynamic customization. In other words, if the compilation were fast enough, dynamic customization could be efficiently supported by dynamically compiling programs after customization.

3.1. Basic Framework

Recently, *run-time specialization* (RTS) techniques, which efficiently perform partial evaluation, have been widely studied[4, 5, 7, 12].

The basic idea of RTS is to construct a *native-code level generating extension* for a given program at compile-time. Let $R^{L\to M}$ be an RTS system from a language L to a machine language M, and p^L be a program being specialized. A *generating extension* G_p, which is constructed by R at compile-time, is a program that takes static parameters of p and generates specialized version of p in M at run-time:

$$\begin{array}{rcll} R^{L\to M}(p^L) & = & G_p & \text{(compile-time)} \\ G_p(x) & = & p_x^M \ . & \text{(run-time)} \end{array} \tag{5}$$

Note that G_p generates specialized program p_x^M in the machine language M, whereas traditional partial evaluation techniques generate in the same language as that the source program is written in. This is done by building fragments of compiled code of p (so called *templates*) at compile time, and by generating specialized programs by merely copying the templates at run-time. As a result, the latter process in RTS systems is highly efficient—it is reported that G_p merely executes a small number of instructions (from a few to a few tens, depending on systems) on average to generate one instruction of p_x^M.

3.2. An Example

Let us see how an RTS system works by taking a simple function `pow`, which computes n'th power of `x` in Scheme, as an input to $R^{L\to M}$:

```
(define (pow n x)
  (if (= n 0)
    1
    (* (pow (- n 1) x))))
```

When we specify that the above function will be specialized with respect to the first argument `n`, the system first determines which expressions can be computed at partial evaluation time (so called *static expressions*), and which expressions shall be inserted in the specialized program (so called *dynamic expressions*). This information is represented as the following *annotated* program:

```
(define (pow n x)
  (if (@ = n 0)
    1
    (@ * (@ pow (@ - n 1) x))))
```

Generating Extension:

```
(define (pow-gen n)
  (load-template t0)
  (if (= n 0)
      (load-template t1)
      (begin (load-template t2)
             (pow-gen (- n 1))
             (load-template t3))))
```

Templates:

label	instructions	comments
t0	push %ebp	// prologue
	movl %esp, %ebp	
t1	movl $1, $eax	// return 1
	leave	// epilogue
t2	push -8(%ebp)	// parameter passing
t3	addl $4, %esp	
	imull -8(%ebp),%eax	// x*pow(x-1)
	leave	// epilogue

Figure 2. Generating extension and templates for `pow`. For readability, instructions in the templates are written in the Intel x86 assembly language. The instructions follow the standard calling convention; a function caller sets its arguments on the stack (accessed via `%esp` in the callee), and receives a return value in `%eax`.

where the underlines and @-marks are *annotations*. An underlined expression is a *dynamic expression*. An expression without underlines is a *static expressions*, whose value can be computed at specialization-time. An @-mark specifies how a function application is processed at partial evaluation. When an @-mark does not have under- or over-lines, the application is executed at partial evaluation time. When it has an underline (*i.e.*, $\underline{@}$), the function application form is inserted in the specialized program. When it has an over-line (*i.e.*, $\overline{@}$), the body of the function is specialized and then inlined at the position.

From the annotated definition, the system creates a generating extension and templates (Figure 2). Basically, the generating extension is created by replacing each dynamic expression with a form that stores an appropriate template into memory (*i.e.*, `(load-template tn)` in the figure). Templates are created by compiling dynamic expressions in the annotated definition. In addition, a prologue and an epilogue instructions are

```
push %ebp       // (pow-gen 3):t0
movl %esp,%ebp
push -8(%ebp)   // (pow-gen 3):t2
push %ebp       // (pow-gen 2):t0
movl %esp,%ebp
push -8(%ebp)   // (pow-gen 2):t2
push %ebp       // (pow-gen 1):t0
movl %esp,%ebp
push -8(%ebp)   // (pow-gen 1):t2
push %ebp       // (pow-gen 0):t0
movl %esp,%ebp
movl $1,%eax    // (pow-gen 0):t1
leave
addl $4,%esp    // (pow-gen 1):t3
imull -8(%ebp),%eax
leave
addl $4,%esp    // (pow-gen 2):t3
imull -8(%ebp),%eax
leave
addl $4,%esp    // (pow-gen 3):t3
imull -8(%ebp),%eax
leave
```

Figure 3. Specialized version of `pow`. A comment like `(pow-gen 3):t0` shows the beginning of a template (`t0`), which is written during an execution of a generating extension (`(pow-gen 3)`).

attached to the first and the last templates, respectively, so that parameters can be passed among inlined templates. The detailed discussion on the creation of generating extensions and templates can be found in the studies on Tempo[4].

When 3 is applied to `pow-gen`, for example, it generates a function that has the instruction sequence shown in Figure 3, which has essentially the same computation to the compiled code of the following Scheme function:

```
(define (pow-3 x) (* x (* x (* x 1)))).
```

3.3. Run-time Specialization as Dynamic Compilation

By using a run-time specializer in place of a partial evaluator in the first Futamura projection (introduced in Section 2.2.3), we can basically compile an interpreted programs at run-time.

Let i^L be an interpreter program written in L, and let p^I be an interpreted program written in I. The

program p^I may be constructed at run-time. A dynamic compilation scheme by using an RTS system is described as follows. At compile-time, we create a generating extension for i by using the RTS system:

$$R^{L \to M}(i^L) = G_i. \quad (6)$$

At run-time, when p^I is to be interpreted by i, we can compile p^I by applying it to G_i and obtain i_p^M, which is a machine language (*i.e.*, compiled) program of p^I:

$$G_i(p^I) = i_p^M \quad . \quad (7)$$

As we are using RTS, i_p^M would be obtained with a sufficiently small amount of overheads.

3.4. Technical Problems

In order to design and implement the above compilation scheme in a reflective language, the following problems should be addressed.

- A meta-level architecture that is appropriate to the above compilation scheme should be designed. In order to dynamically compile programs in the above scheme, we have to create a generating extension for each meta-interpreter at compile-time. This means that the system requires definitions of meta-interpreters before executing a base-level program. (On the contrary, base-level programs can be dynamically modified or constructed, as the system can dynamically compile them.) Therefore, the meta-level architecture, which allows customization of interpreters from application programs, should have mechanisms (in other words, restrictions) that make definitions of meta-interpreters known at compile-time.

- The performance of the system including efficiency of compilation process and efficiency of compiled programs should be examined. As for the efficiency of compiled programs, compile-time (or traditional) partial evaluation and run-time specializations basically generate specialized programs in the same quality, because both can remove the same expressions (*i.e.*, static expressions) from the original programs. RTS-specialized programs, however, have a certain amount of overheads. For example, the specialized program in the previous section (Figure 3) has a considerable amount of overheads in passing parameters among inlined templates. Nöel et al. showed that there are 10–30% overheads in RTS generated programs[19]. In addition, the overheads in compiled reflective languages are not known. Specialization speed (*i.e.*, compilation speed) should also be examined.

Our approach to these problems is to implement a prototype system, and examine the expressiveness and performance of the system. In the following sections, we first present our simple reflective language, which is implemented by using our RTS system, and we then show performance of the system.

4. A Simple Reflective Language

4.1. Reflective Architecture

Our reflective architecture consists of the base-level and the meta-level (*i.e.*, no infinite tower of meta-levels). A user-program separately defines meta-level functions and base-level functions.

The meta-level part of a program has more than one meta-interpreters, each of which is defined by the following form:

`(define-eval (`*name e r k s*`) body)`,

where the *name* is the name of the interpreter, and *e*, *r*, *k* and *s* are current expression, environment, continuation and store of an executing program, respectively. The semantics of the form is not different from that of the standard `define` form except for the *name* is registered in a table so that it can be used by the base-level program.

The base-level part of a program can use two special constructs in addition to a functional part of constructs in Scheme. The construct `(get-eval` *name*`)` looks up a meta-interpreter that is defined by the given *name*, and returns a reference to the meta-interpreter. The construct `(eval-with` *int exp*`)` evaluates an expression *exp* by using a meta-interpreter *int*. The parameter *int* must be a reference to a meta-interpreter.

In a sample program shown in Figure 4, two meta-interpreters `verbose` and `default` are defined at the meta-level. At the base-level, the `default` interpreter is selected to evaluate programs by default. In function `sum-of-numbers`, one of the two meta-interpreters is selected according to a dynamic condition, and an expression is evaluated under the selected meta-interpreter. Note that the second parameter to `eval-with` is quoted. This is because the form can evaluate expressions that are constructed at run-time.

4.2. Implementation Sketch

Under the compilation scheme introduced in Section 3.3, we use an RTS system to dynamically compile

```
;;; meta-level definition
(meta
 ;; a verbose interpreter that prints each
 ;; expression being evaluated, and the
 ;; result of the evaluation.
 (define-eval (verbose e r k s)
   (display e) (newline)
   (let ((k* (lambda (a) (display "result=")
               (display a) (newline)
               (k a))))
     (cond ((constant? e) (k* e))
           ((variable? e) (k* (s (r e))))
           ...)))
 ;; a standard interpreter
 (define-eval (default e r k s)
   (cond ((constant? e) (k e))
         ((variable? e) (k (s (r e))))
         ...)))

;;; base-level definition
(base 'default ; specify default interpreter
  (define (sum-of-numbers verbose? n)
    (eval-with ;; selection of a
               ;; meta-interpreter
               (if verbose?
                (get-eval 'verbose)
                (get-eval 'default))
     ;; the following expression is executed
     ;; by the selected interpreter
     '(letrec ((sum
                (lambda (n)
                  (if (zero? n)
                    0
                    (+ n (sum (- n 1)))))))
        (sum n)))))
```

Figure 4. A Sample Program that Dynamically Selects a Meta-Interpreter.

```
;; evaluation of (eval-with int exp)
(define (eval-eval-with e r k s0)
  (eval (cadr e) r      ; evaluate int
    (lambda (int s1)
      (eval (caddr e) r ; evaluate exp
        (lambda (exp s2)
          (eval-with-body int exp r k s2))
        s1))
     s0))
;; body of eval-with
(define (eval-with-body int exp r k s)
  (let* ((cache (get-cache int))
         (key (list exp r))
         ;; lookup precompiled code
         (code (lookup cache key)))
    (if code
        ;; execute precompiled code
        (apply k (code s))
        (let ((code*  ; compile exp
               ((get-specializer int) exp r
                (lambda (x s*)
                  (list x s*)))))
          ;; register the compiled code
          (set-hash! cache key code*)
          (apply k (code* s))))))  ; exec.
```

Figure 5. Implementation of `eval-with`.

base-level programs that are executed by customized interpreters. Basic strategy is to construct a generating extension for each meta-interpreter definition at compile-time, and when an `eval-with` form is executed at run-time, apply an expression in the form to the generating extension.

For each meta-interpreter definition (`define-eval` (*name e r k s*) *body*), the system constructs a pair of a generating extension of the function and a hash table that records compiled code. The generating extension assumes that the parameters *e*, *r*, and *k* are static.

When a (`eval-with` *int exp*) form is evaluated, it first evaluates sub-expressions *int* and *exp* under the current interpreter. It then looks up the value of *exp* in the hash table of *int*. If there is a code in the table, it simply executes the code. Otherwise, it creates a compiled code by invoking the generating extension of *int*, then put the compiled code in the hash table, and finally executes the code with dynamic parameter *s*. The behavior of `eval-with` is presented in Figure 5.

Our current implementation uses our own RTS system, which applies Tempo's template mechanisms[4] to Scheme. To build templates of Scheme expressions,

specialization	application: matrix time	matrix speedup	deriv time	deriv speedup
none	22.5	(1.0)	8.06	(1.0)
run-time	4.81	4.68	1.54	5.23
compile-time	4.00	5.63	1.20	6.69
overhead (RT/CT)	20.2%		27.9%	

Table 1. Performance of compiled programs. (Execution times are displayed in milliseconds.)

speciali-zation	application: matrix time	matrix BEP	deriv time	deriv BEP
run-time	5.0	0.28	6.0	0.92
compile-time	10.5×10^3	594	14.7×10^3	2254
speedup (CT/RT)	2.1×10^3		2.5×10^3	

Table 2. Compilation times and their break-even points. (Compilation times are displayed in milliseconds.)

we customized DEC Scheme-to-C compiler and lcc, a portable C compiler. Current system targets the Intel x86 architecture.

5. Performance Measurements

We measured performance of our prototype system with simple meta- and base-level programs. We wrote an interpreter of a small subset of Scheme as a meta-level program, and two functions (matrix multiplication 'matrix' and differentiation of symbolic expression 'deriv') as base-level programs. The interpreter was executed in the following ways:

none: The interpreter was simply compiled and executed (*i.e.,* interpreted) the base-level programs.

run-time: The interpreter was specialized for each base-level program by using our run-time specialization system, and the generated (*i.e.,* dynamically compiled) code was executed.

compile-time: The interpreter was specialized by using Similix, a Scheme partial evaluator, and the specialized (*i.e.,* statically compiled) code was executed after compiling by using Scheme-to-C compiler.

The programs were executed on a PC-compatible with 150MHz Pentium processor and 48MB memory.

Criteria of the measurement are (1) overheads in compiled code, and (2) speed of compilation.

5.1. Overheads in Compiled Code

Table 1 shows the execution times and speed-up ratios to the interpreted executions. The 'time' columns show execution times of base-level programs (excluding times spent for compilation, even in run-time specialized executions). The 'speedup' columns show the factors of reduced execution times of specialized (or compiled) executions from the interpreted executions. The bottom row shows the overheads in executions of dynamically specialized programs, compared to executions of statically specialized programs.

The results showed that our RTS system generates efficient specialized meta-interpreters, namely more than four times faster than the interpreted executions. Note that dynamically compiled programs yet have a certain amount of overheads (20–30%), compared to statically compiled executions. This will be discussed in Section 6.

5.2. Compilation Speed

Table 2 shows times spend for compiling base-level programs by using run-time and compile-time specialization techniques. The 'time' columns show the elapsed times for obtaining executable native programs for each base-level program. In the compile-time specialization, the figures include the times for compiling specialized source programs into native programs although the specialization times were dominant in our experiments (we specialized and compiled using Similix and Scheme-to-C, respectively). Thanks to our RTS system, the times for compilation processes are reduced by three orders of magnitude.

The 'BEP' columns show the *break-even points* of specialized programs. A break-even point of a specialization process is the number of runs of specialized program that amortizes the time of the specialization process, which can be computed by the following formula:

$$p/(o - s),$$

where p, o and s are the time for specializing the original program, the time for executing the original program and the time for executing the specialized program, respectively. In the cases of RTS, the break-even points are less than one because the reduced execution

times by the specialization are greater than the times of the specialization (compilation)[1].

6. Approaches to Reduce Overheads

As we have seen, dynamically compiled base-level programs still have a certain amount of overheads. Below, we discuss the sources of the overheads, and our approaches to reduce them.

6.1. Sources of Overheads

In terms of the performance of specialized programs, one of the crucial differences between run-time and compile-time specialization techniques exists in function inlining. Although the run-time specialization techniques can perform function inlining during specialization, their specialized code has a certain amount of overheads in between inlined functions. The reasons can be explained as follows. As we have introduced in Section 3, an RTS system basically creates templates (fragments of native machine instructions) by compiling dynamic expressions in an original program. When a function invocation is inlined during specialization, the caller's and the callee's templates are consecutively copied into the memory. Those templates are created and compiled per-function basis; a template is created without knowing what function will be inlined together. Consequently, the specialized program have to save and restore registers between inlined templates so that caller's registers are preserved after the execution of callee's templates. This operation costs roughly the same execution time to a function invocation. In addition, peep-hole optimizations, such as instruction scheduling, can not be performed across those inline templates.

In meta-interpreters, the amount of overheads inserted in between inlined templates become relatively larger than typical benchmark programs of RTS, because of templates of meta-interpreters, which correspond to each expression type in base-level programs, tend to be small. Let us see this problem by an example. Assume that a meta-interpreter shown in Figure 6 evaluates an expression `(eq? x 1)` in a base-level program. When an RTS system specializes the interpreter with respect to `(eq? x 1)`, it copies three templates,

[1] In most RTS studies, the BEPs in their benchmark programs are more than one because they usually measure the execution time of a small fragment of computation (so called kernels). In our programs, a loop in a base-level program repetitively executes a fragment of a specialized code, the specialization time for the loop is relatively smaller.

```
(define (eval exp env store)
  (cond ...
    ((and (@ pair? exp)
          (@ eq? (@ car exp) 'eq?))
     (@ eq? (@̄ eval (@ cadr exp) env store)
            (@̄ eval (@ caddr exp) env store)))
    ...))
```

Figure 6. Semantics `(eq?` e_1 e_2`)` **in an Annotated Interpreter Definition. In the definition, continuations are made implicit for readability.**

namely variable reference (by inlining the first recursive invocation of `eval`), constant value (by inlining the second), and equality test (the intrinsic template for `eq?`). This means that the base-level programs compiled by using an RTS system have 'function-invocation' overheads for each sub-expression, even for a constant value or a variable reference!

6.2. Speculative Function Inlining

One of our approaches to alleviating this problem is to *statically* inline recursive invocations of meta-interpreters so that it reduces the number of dynamically inlined invocations. For example, if we statically inlined the code for variable reference, we could have a meta-interpreter definition shown in Figure 7. Variable references in the dynamically specialized version of this interpreter can be achieved without paying 'function-invocation' overheads between inlined templates.

For the time being, we are examining the ways to appropriately inline such recursive invocations. Of course, inlining techniques could easily result in code explosion. Our approach would therefore inline programs only for specific patterns of base-level expressions.

6.3. Optimizations after Specialization

Another approach to reduce the overheads between inlined templates is to perform optimization *after specialization*. Since most of existing RTS systems generate specialized programs by merely copying precompiled native machine instructions into memory, there are no chance to optimize among inlined templates.

From the above observation, we are currently studying an RTS system that generates specialized programs in an intermediate language, and then quickly com-

```
(define (eval exp env store)
  (cond ...
    ((and (@ pair? exp)
          (@ eq? (@ car exp) 'eq?))
     (@ eq?
      (let ((exp (@ cadr exp)))
        (if (@ variable-ref? exp)
          do variable reference
          (@ eval exp env store)))
      (let ((exp (@ caddr exp)))
        (if (@ variable-ref? exp)
          do variable reference
          (@ eval exp env store))))
        ...))
```

Figure 7. Speculatively Inlined Variable References in an Interpreter.

piles them into a native machine language. By performing fast yet effective optimization techniques in the latter process, we could obtain efficient specialized programs. Our experimental system generates specialized programs in the Java virtual machine language, thus enables to use sophisticated just-in-time compilers. Thus far, dynamically specialized programs generated by our experimental system showed closer performance to the statically specialized counterparts[18]. Our future plan is to fully implement our RTS system, and apply the system to meta-interpreters.

7. Conclusion

In this paper, we presented dynamic compilation techniques of reflective languages. Since reflective languages can be regarded as the system with mechanisms for dynamic software evolution, we believe that studies on implementation techniques of reflective languages would also be applicable to other systems with mechanisms for dynamic software evolution in principle.

As a 'proof-of-concept' of our approach, we presented a simple reflective architecture that allow application programs to statically define customized meta-interpreters, to dynamically construct base-level programs, and to dynamically select meta-interpreters. We also showed an dynamic compilation scheme by using a run-time specialization technique.

We implemented an experimental system based on the above architecture and compilation scheme, and measured basic performance of the system. Our benchmark programs showed that (1) dynamically compiled base-level programs run more than four times faster than the interpreted ones, (2) they yet have 20–30% overheads over statically compiled programs, and (3) dynamic compilation process is so efficient that the cost can be easily amortized in a few runs of compiled programs. This paper also discussed sources of overheads in dynamically compiled base-level programs, and discussed several approaches to reduce them.

References

[1] N. Amano and T. Watanabe. A procedural model of dynamic adaptability and its description language. In Katayama [10], pages 103–107.

[2] K. Asai, S. Matsuoka, and A. Yonezawa. Duplication and partial evaluation —for a better understanding of reflective languages—. *Lisp and Symbolic Computation*, 9:203–241, 1996.

[3] S. Chiba. A metaobject protocol for C++. In Loomis [13], pages 285–299.

[4] C. Consel and F. Noël. A general approach for runtime specialization and its application to C. In *Conference Record of Symposium on Principles of Programming Languages (POPL96)*, pages 145–170, St. Petersburg Beach, Florida, Jan. 1996. ACM SIGPLAN-SIGACT.

[5] N. Fujinami. Automatic and efficient run-time code generation using object-oriented languages. *Computer Software*, 15(5):25–37, Sept. 1998. (In Japanese).

[6] Y. Futamura. Partial evaluation of computation process—an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[7] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Annotation-directed run-time specialization in C. In *Proceedings of Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, volume 32(12) of *ACM SIGPLAN*, pages 163–178, Amsterdam, June 1997. ACM.

[8] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, H. Tezuka, H. Konaka, M. Maeda, and K. Kubota. Design and implementation of metalevel architecture in C++: MPC++ approach. In G. Kiczales, editor, *Reflection'96*, pages 153–166, San Francisco, California, Apr. 1996. Proceedings are available at http://jerry.cs.uiuc.edu/reflection/reflection96/index.html.

[9] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[10] T. Katayama, editor. *International Workshop on Priciples of Software Evolution (IWPSE'98)*, Kyoto, Japan, Apr. 1998.

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

[12] M. Leone and P. Lee. Lightweight run-time code generation. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, Orlando, FL, June 1994. ACM SIGPLAN. published as Technical

Report 94/9, Department of Computer Science, The University of Melbourne.

[13] M. E. S. Loomis, editor. *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, volume 30(10) of *ACM SIGPLAN Notices*, Austin, TX, Oct. 1995. ACM.

[14] P. Maes. Concepts and experiments in computational reflection. In N. Meyrowitz, editor, *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, volume 22(12) of *ACM SIGPLAN Notices*, pages 147–155, Orlando, FL, Oct. 1987. ACM, ACM.

[15] H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In Loomis [13], pages 300–315.

[16] H. Masuhara and A. Yonezawa. Design and partial evaluation of meta-objects for a concurrent reflective language. In E. Jul, editor, *European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *Lecture Notes in Computer Science*, pages 418–439, Brussels, Belgium, July 1998. Springer-Verlag.

[17] H. Masuhara and A. Yonezawa. A reflective approach to support software evolution. In Katayama [10], pages 135–139.

[18] H. Masuhara and A. Yonezawa. Generating optimized residual code in run-time specialization. In *Proceedings of International Colloquium on Partial Evaluation and Program Transformation*, pages 83–102, Waseda University, Tokyo, Japan, Nov. 1999.

[19] F. Noël, L. Hornof, C. Consel, and J. L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *IEEE International Conference on Computer Languages (ICCL '98)*, pages 123–142, Chicago, Illinois, USA, May 1998.

[20] H. Okamura and Y. Ishikawa. Object location control using meta-level programming. In M. Tokoro and R. Pareschi, editors, *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 821 of *Lecture Notes in Computer Science*, pages 299–319. Springer-Verlag, July 1994.

[21] B. C. Smith. Reflection and semantics in Lisp. In *Conference record of Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[22] M. Wand and D. P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In P. Maes and D. Nardi, editors, *Meta-Level Architecture and Reflection*, pages 111–134. Elsevier Science, North-Holland, 1988.

[23] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of OOPSLA'88 (SIGPLAN Notices Vol.23, No.11)*, pages 306–315, San Diego, CA, Sept. 1988. ACM. (revised version in [24]).

[24] A. Yonezawa, editor. *ABCL: An Object-Oriented Concurrent System*. MIT Press, Cambridge, MA, 1990.