Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation

Hidehiko Masuhara* **Satoshi Matsuoka†** **Kenichi Asai** **Akinori Yonezawa**

Department of Information Science, University of Tokyo

†Department of Information Engineering, University of Tokyo

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

E-mail: {masuhara, matsu, asai, yonezawa}@is.s.u-tokyo.ac.jp

Abstract

Meta-level programmability is beneficial for parallel/distributed object-oriented computing to improve performance, etc. The major problem, however, is *interpretation overhead* due to meta-circular interpretation. To solve this problem, we propose a compilation framework for object-oriented concurrent reflective languages using partial evaluation. Since traditional partial evaluators do not allow us to directly deal with meta-circular interpreters written with concurrent objects, we devised techniques such as pre-/post-processing, a new proposed *pre-action* extension to partial evaluation in order to handle side-effects, etc. Benchmarks of a prototype compiler for our language ABCL/R3 indicate that (1) the meta-level interpretation is essentially 'compiled away,' and (2) meta-level optimizations in a parallel application, running on a Fujitsu MPP AP1000, exhibits only 10–30% overhead compared to the hand-crafted source-level optimization in a non-reflective language.

1 Introduction

In parallel/distributed object-oriented applications, *meta-level programming*, such as load balancing, data allocation, scheduling, etc., is greatly beneficial. Language extensions that provide modular abstractions for such meta-level programming are also useful. Ability to provide meta-level programmability and language extensibility is achieved via *Open Implementation* languages, or more traditionally, *reflective languages* [5, 12, 14, 17]. Thanks to the success of CLOS Metaobject Protocol (MOP)[12], reflective languages are now regarded as practical tools rather than mere philosophical toys.

The two foremost requirements in reflective languages are *good programmability* and *efficiency*. The former is obvious for a variety of reasons, ranging from reducing the cost of meta-level programming to increasing safety. The latter is also important, since one of the primary purpose of meta-level programming is performance improvement. Let us analyze the two major approaches to language design in this regard, *meta-level (reflective) interpreters* and *customizable compilers*:

Meta-level (Reflective) Interpreters: The traditional approach in reflective languages is to construct a meta-level interpreter, and provide reification/reflection interfaces. This approach gives the programmer a clear and concise execution model of the language, helping him to obtain the confidence that his meta-level programming 'correctly' implements his intentions. Furthermore, object-oriented factorization of interpreters into meta-objects to form a meta-level class framework as pioneered in CLOS-MOP, al-

*JSPS Research Fellow.

To appear in OOPSLA'95.

low modular and incremental programming and scope control. Almost all successful reflective languages to date have adopted this design.

Unfortunately, the major problem of this approach is efficiency. The overhead of run-time interpretation, if implemented naively, easily becomes a factor of 10 up to 1000 over optimized, directly compiled execution. Such performance penalty is unacceptable especially in parallel/distributed applications. Past solutions to the problem fall into three inter-related categories:

- Some parts of the meta-system are not subject to meta-level modification via reification. For example, in the reflective language Open C++[5], only message passing, object creation, and instance variable accesses can be reified. This not only restricts user programmability, but also makes the language model unclear, in that much of the meta-level functionalities are hidden inside 'black-boxes,' and the programmer may not obtain a clear view of how his meta-level programming and the black boxes will interact.
- The system embodies a set of ad-hoc optimizations transparent to the user. For example, our previous work ABCL/R2[14] assumes that most objects will not be customized, and thus could be mostly compiled, and switched to general interpreted execution upon user customization. Its effectiveness is limited to, however, cases when optimization is possible, and interpretation overhead heavily impacts the overall performance otherwise.
- Applications where algorithmic performance improvement overcomes the cost of interpretation are looked for, as was experimented with AL-1/D[16]. Although a sound approach, this restricts the application of open implementation technology, as it is unlikely that such dramatic algorithmic improvements are always possible. (For more detailed discussion, see Section 5.)

Although most reflective languages integrate all the above approaches, none runs efficiently under all circumstances.

Customizable (Open) Compilers: Opening-up the internal structure of a compiler to user modifications is another way to allow meta-level programming and language customization. Because customization is done at compile-time, there is no run-time overhead of meta-level interpretation. Several researchers have observed that providing object-oriented interface to the compiler (in the spirit of Metaobject Protocol) lessens the customizations effort[13, 18].

Despite such efforts, programmability of this approach cannot be regarded as good at present. It is often not obvious to the programmer whether his compiler customization correctly captures his intended meta-level functionality. This is due to several reasons, including (1) since the size of a typical compiler is enormous, it is difficult for the user to grasp its structure even with object-oriented techniques, (2) because of the way compilation works, it is difficult to localize or to properly propagate the changes made, and (3) the programmer has to constantly distinguish the values handled in his customization as being either compile-time or run-time values[1].

We have developed a new compilation framework where we can enjoy the fruit of both approaches in object-oriented concurrent reflective languages. In our framework, the meta-level of the language is exposed to the programmer as a *pure* meta-circular interpreter organized in an object-oriented way, as is with traditional approaches. Then, the interpretation overhead is effectively eliminated by the compiler, i.e., the meta-level interpreter is 'compiled away.' Our compilation framework is based on *partial evaluation*[2], or more specifically, is a non-trivial realization of the first Futamura projection[7]. Of course, simple application of conventional partial evaluation technology does not work. The techniques we have developed are as follows:

[1] This is similar in principle to Lisp macros, but much more complicated to handle.

- *Pre-processing* that converts object-oriented meta-circular evaluator definitions into *continuation passing style* (CPS) functions, so that they become applicable to partial evaluation.

- An extension to partial evaluation called *preactions*, which preserves the order and the number of *I/O side effects*.

- *Multiple compiled methods* for dynamic modification of the meta-level.

- Proper treatment of assignment side-effects.

- *Post-processing* that further optimizes the partially evaluated programs before passing it on to the back-end compiler.

We have developed a prototype compiler for an object-oriented concurrent reflective language ABCL/R3 according to our compilation framework. Preliminary benchmarks show that: (1) interpretation overhead is effectively eliminated, i.e., the programs compiled by our compiler exhibit almost identical performance to the ones compiled by non-reflective compilers, and is more than 100 times faster compared to interpreter execution; and (2) parallel applications on a massively parallel processor Fujitsu AP1000 optimized via meta-level programming adds only small overhead compared to hand-crafted source-level optimizations, and runs *faster* than non-optimized base-level programs compiled by a non-reflective compiler. This facilitates creation of a portable, meta-level class framework for optimization and language extensions.

The rest of this paper is organized as follows. Section 2 gives a simple example of an ABCL/R3 program, and an overview of our compilation framework. Section 3 describes the techniques of our compilation framework in detail. Section 4 shows results of our preliminary benchmarks. Section 5 discusses related work, and Section 6 concludes the paper.

2 Compiling Away the Meta-Level: an Overview

2.1 A Simple Compilation Example in ABCL/R3

We give an overview of our compilation framework with a simple meta-level programming example shown in Figure 1. The base-level program is a fragment of a client-server system where the server assigns a worker object to a client object according to the estimated cost of processing the clients' request. The meta-level embodies a simple tracing system over some variable references. A method `request` (the program ① in the figure) of the object `server` asks each worker object the estimated cost and returns the best one to the object `client`[2]. Here, we customize the meta-level interpreter of the base-level algorithm so that when variables `worker2` and `client` are referenced, the reference events are reported to the object `*console*` by messages `notify`.

In ABCL/R3, the default meta-circular interpreter is defined as methods of the primary evaluator object `prim-eval` (the program ② in the figure). The above customization is achieved by defining a new evaluator object `watch-eval` to override the method `eval-var` that defines the behavior of variable references (the program ③ in the figure). The method sends a notification to the object `*console*` if the name of the referenced variable matches `worker2` or `client`. The execution of all the other expressions are *delegated* to the object `prim-eval`.

Since the customization changes the semantics of the language from the original ABCL/R3, a naive implementation has to execute the compiled base-level program under the customized meta-interpreter, instead of directly executing the base-level program. This execution is more than 100-times slower as we show in Section 4.1. However, since the names of variables that are introspected can be statically determined, it would be enough

[2]The source program is written in ABCL/R3, based on the parallel object-oriented language ABCL/f[23]. A message send is written as a method invocation form: (⟨*method*⟩ ⟨*target-object*⟩ ⟨*arguments*⟩...).

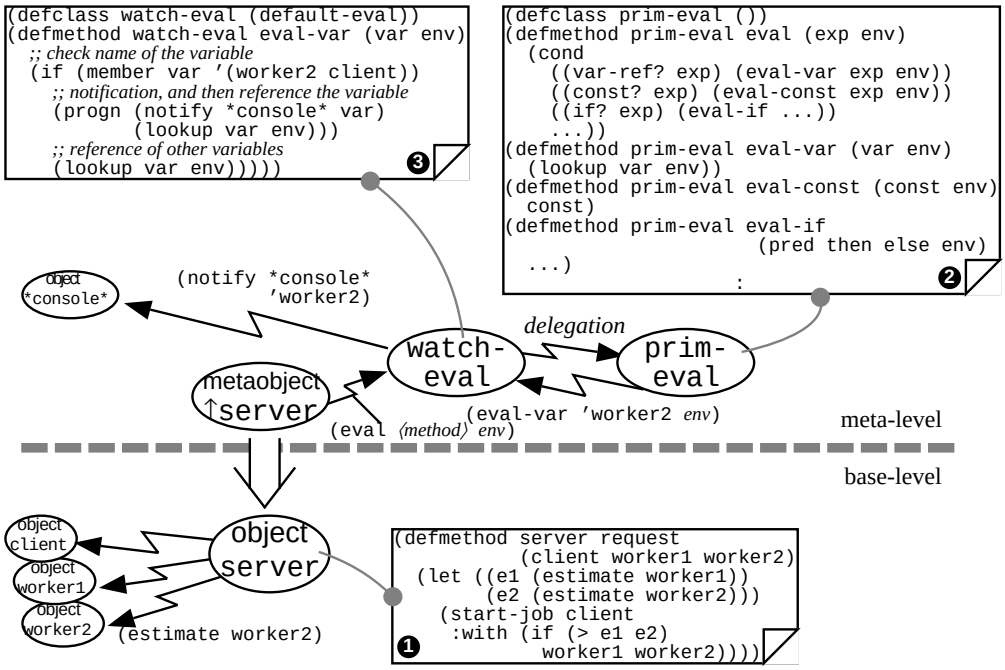


Figure 1: A Meta-Level Programming Example in ABCL/R3

```
(defmethod server request
                    (client worker1 worker2)
 (let ((t313 (estimate worker1)))
   (notify *console* 'worker2) ;; notification
   (let ((t314 (estimate worker2)))
     (notify *console* 'client) ;; notification
     (if (> t313 t314)
       (start-job client :with worker1)
       (progn ;; notification
         (notify *console* 'worker2)
         (start-job client :with worker2))))))
```

Notifications of underlined variables are inlined into the base-level program.

Figure 2: Code Generated by Our Compiler

to insert notification code into specific variable accesses in the base-level program[3]. In fact, our compiler generates exactly such an efficient program (shown in Figure 2) by 'compiling away' all the unnecessary interpretation.

As mentioned earlier, our compiler is mainly based on partial evaluation[10] to eliminate meta-level interpretation. The readers should note that traditional inlining optimization techniques such as the ones in Self[4] would have replicated almost *the entire* interpreter code, instead of the notification code. Thus, partial evaluation is a quite essential part of the compilation process. However, simply applying traditional partial evaluation techniques is insufficient. In the rest of this section, we present the basic idea of compiling reflective programs using partial evaluation, and the problems when applied to concurrent objects.

2.2 Compiling Reflective Programs Using Partial Evaluation

A partial evaluator PE is a function that takes a program P and specification s (partial information on P's input), and returns a specialized program P_s (called the *residual* program). For brevity, we assume that P takes two arguments, and the specification is given as the first argument. For program P and value x as its first argument, we denote a specialized version of P as $P_x = PE(P, x)$. The special-

[3]By all means, having the programmer do so manually throughout the entire program would be quite cumbersome.

ized program satisfies the condition $P(x,y) = P_x(y)$ whenever $P(x,y)$ has an answer.

Now we model a reflective system; let I be a default meta-level interpreter that takes a base-level program B and run-time data D as arguments. Since the system is reflective, the user can execute B not only under the original interpreter I, but also under a modified interpreter I'.[4]

Under the interpreter I, B can be "directly executed" by $B(D)$, since $I(B,D) = B(D)$. Once the interpreter is modified to be I', however, previous reflective systems execute B by "interpretation" $I'(B,D)$, which is significantly slower. Now, given a partial evaluator, we partially evaluate I' with respect to B, yielding I'_B; i.e., $I'_B = PE(I',B)$. If we can expect that the partial evaluator is powerful enough to execute all 'interpretations' in I' at partial evaluation time, I'_B will be almost the same as B except that the effect of modifications on I' are embedded and other parts of the interpreter are 'compiled away.' Consequently, the execution of $I'_B(D)$ will have the same result as that of $I'(B,D)$, being as fast as that of $B(D)$, or even faster if modification on I' was made to improve performance.

2.3 Problems in existing Partial Evaluation Techniques when Applied to Concurrent Objects

Unfortunately, the above scheme is an ideal case. In practice, existing partial evaluation techniques do not allow us to directly deal with the meta-circular interpreters written in concurrent object-oriented languages. Here we explain the underlying problems and our proposed solutions.

Concurrent meta-system.
As the previous studies show[14, 17, 24], it is natural to design the meta-system of concurrent object-oriented languages with concurrent objects. However, it is difficult to eliminate the meta-level interpretation by partially evaluating the *entire* meta-system, because of the concurrency and indeterminacy of concurrent objects. To the best of our knowledge, partial evaluation studies that deal with meta-circular interpreters assume functional languages.

Solution: The meta-system of ABCL/R3 is designed with concurrent objects, and converted into CPS (continuation passing style) functions before partial evaluation. The partial evaluator specializes the converted functions for each base-level method; i.e., only the body of method execution is the target of partial evaluation. This means that we could lose the opportunity to specialize some interactions among objects, but allows the application of most existing partial evaluation techniques.

Side-effect in programs. There are two types of side-effects in concurrent object-oriented languages, (1) interaction among objects (e.g., message passing and synchronization) and (2) assignment to instance variables of an object. Simple partial evaluators cannot correctly translate programs with such side-effects. Operations with side-effects would be duplicated, disappear, or may appear in different order to the original program.

Solution: For (1), we propose a partial evaluation mechanism called *preaction* for preserving the characteristics of object interactions. With this mechanism, the number and the order of operations in the interaction are preserved after partial evaluation. As for (2), we convert the meta-interpreter into store passing style, so that assignment can be represented without using side-effects at the meta-level. Store operations in the residual program are eventually converted into variable update forms by post-processing.

Dynamic modification of the Meta-system.
Most reflective systems allow dynamic alteration of the meta-system. If the meta-interpreter definition admits this flexibility, the partial evaluator can not eliminate the interpretation as we expected in Section 2.2, because we cannot statically determine what program is actually being

[4] Here, we assume that the definition of I' is statically known. ABCL/R3 does allow dynamic modification of the meta-interpreters to some extent. It is not achieved by partial evaluation, but by multiple compiled methods as explained in Section 3.2.3.

executed at partial evaluation time.

Solution: Our approach is to employ code versioning; i.e., statically generate *multiple compiled methods* for a single base-level method for all possible meta-interpreter definitions used at run-time. In ABCL/R3, we design the language so that modifications made to the meta-interpreter is achieved by introducing new evaluator objects; as a result, the meta-interpreter definitions which could be used at run-time are easily determined at compile-time.

3 Our Compilation Scheme

We have developed a prototype compiler for the ABCL/R3 system. To implement the above solutions, we divide the compilation into four phases (Figure 3) each of which performs the following: ① *pre-processing*: the evaluator object definitions (including user defined ones) are converted into a set of CPS functions to be partially evaluated; ② *partial evaluation*: the converted functions are specialized with respect to each base-level method, yielding a set of residual programs; ③ *post-processing*: the residual programs are optimized into an efficient ABCL/*f* program[23]; and ④ *back-end compilation*: the generated program is compiled into an executable code by the ABCL/*f* compiler. The following sections describe each step in detail.

3.1 Pre-Processing: Conversion from Evaluator Objects into Composed CPS Functions

In ABCL/R3, the evaluator objects which 'interpret' the base-level methods are defined in *direct style* (i.e., directly returns the results of a method call to the caller), and uses *delegation* for incremental customization. Due to certain technical reasons, the partial evaluator expects programs to be in CPS (i.e., passes on the result to the continuation)[5]. In addition, dynamic delegation cannot be well-handled by the partial evaluator, because of loss of static information. To solve such problems, the object definitions are converted as follows in the *pre-processing* phase:

(a) Evaluator methods before conversion

```
;;; A method of class verbose-eval
(defmethod verbose-eval eval (exp env)
  (print exp)
  (delegate)) ; delegation to the next evaluator
;;; A method of class prim-eval
(defmethod prim-eval eval (exp env)
  ... (eval ...) ...) ; call to the head of delegation
```

(b) Converted CPS functions

```
(defun verbose-eval-eval (exp env cont)
  (print exp)
  (prim-eval-eval exp env cont))
(defun prim-eval-eval (exp env cont)
  ... (verbose-eval-eval ...) ...)
```

Figure 4: Pre-process on Evaluator Methods

1. When compiling a base-level method, a set of evaluator objects that execute the method is firstly determined via static analysis.

2. Each evaluator method is converted into a function with a unique name.

3. Functions are converted into CPS.

4. Delegation forms are statically resolved and inlined; each delegation form is substituted with calls to the delegatee's function.

Figure 4 shows an example of pre-processing of an evaluator object `verbose-eval`, which prints each expression given to it, and then delegates the expression to the primary evaluator `prim-eval` for execution. The methods shown in Figure 4(a) are converted into functions shown in Figure 4(b).

3.2 Partially Evaluating Meta-Level Code

Next, the pre-processed meta-level program is specialized with respect to each base-level method using a partial evaluator. We have constructed an online partial evaluator for ABCL/R3 by incorporating our proposed techniques below, in addition to the existing ones[10, 20, 25] such as graph representation of symbolic values and arity raising.

[5]There are more elaborate partial evaluation techniques that allow direct-style interpreters, but we did not employ it for simplicity.

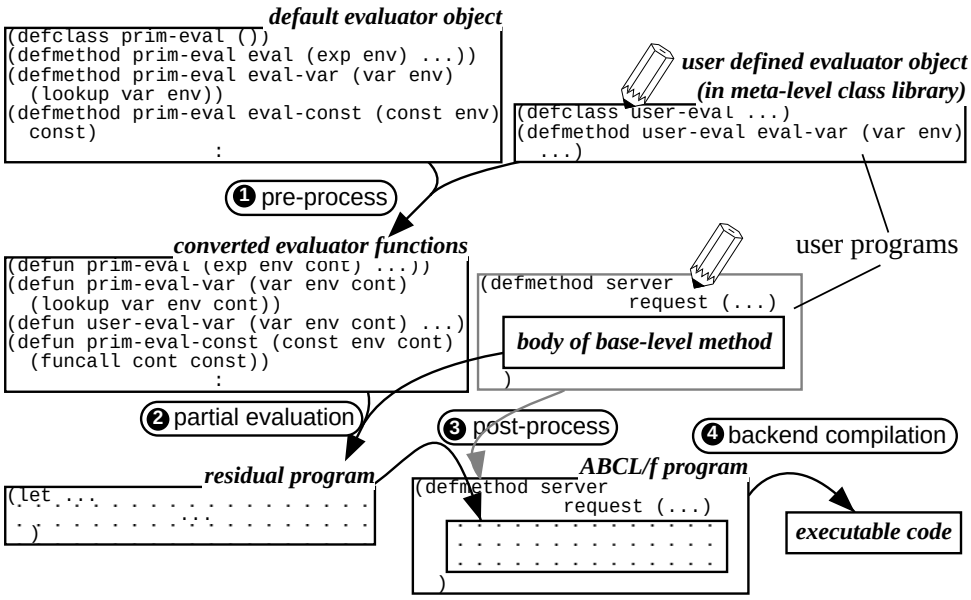


Figure 3: Compilation Phases of ABCL/R3

3.2.1 Handling Side-Effects (1): Object Interactions

Base-level concurrent object-oriented programs involve I/O operations such as message passing, synchronization among objects, etc., that are different from side-effects caused by assignments. (Hereafter, we will refer to these side-effects as the *I/O type side-effects,* as opposed to the side-effects by assignments.) It might seem that such operations could be merely treated as function calls that are executed at run-time (i.e., not subject to unfolding during partial evaluation) by simply extending a partial evaluator for functional languages[6]. However, such a partial evaluator may move or duplicate operations during its execution, and as a result, I/O operations may be eliminated or duplicated, or may appear in a different order to the original one in the residual program(Figure 5).

To solve this problem, we have devised a technique called *preaction*, which properly preserves the number and the order of I/O operations in partial evaluation[2]. A preaction of a symbolic value can be regarded as a history of I/O operations that should be performed before the use of the value. For example, the value of a form:

```
(progn (send x :hello) 123)
```

is 123, but the action `(send x :hello)` should be performed before the value is returned. In our partial evaluator, such a value is represented as:

$$\langle\!\langle \texttt{(send x :hello)} \rangle\!\rangle_{123}$$

(1) Disappearance
`(* 5 (progn (send a :foo) 2))` $\Longrightarrow$ `10`
(2) Wrong Ordering
```
(let ((x (send a :first)))
  (cons (send a :second) x))
```
$\Longrightarrow$ `(cons (send a :second) (send a :first))`
(3) Duplication
`(let ((x (send a :foo))) (cons x x))`
$\Longrightarrow$ `(cons (send a :foo) (send a :foo))`

Figure 5: Examples that I/O side-effects are not properly preserved

[6]Some partial evaluators for imperative languages are reported to be capable of handling side-effects[1, 3, 15]. However, whether they will be effective for our purpose (i.e., whether meta-interpreters are eliminated) is unknown at this time.

The partial evaluation rules for ABCL/R3 extended with preactions is shown in Figure 6. Roughly speaking, the preactions in the arguments of an operator are copied to the ones of their results.

Our partial evaluator uses a graph (DAG) structure to represent symbolic values, in order to preserve the number of I/O operations[2]. When a certain value is used by multiple expressions, it is shared in the graph structure during partial evaluation. At the final phase of the partial evaluation, the shared nodes in the graph are converted to let-forms so that the sharing could be expressed as references to the let-bound variables in the let-body[7].

An example partial evaluation with preactions proceeds as follows:

$$
\begin{aligned}
&\mathcal{PE}[\![\texttt{(* 5 (progn (send a :foo) 2))}]\!]\rho \\
&= \text{apply}(*,\ \mathcal{PE}[\![5]\!]\rho, \\
&\qquad \mathcal{PE}[\![\texttt{(progn (send a :foo) 2)}]\!]\rho) \\
&= \text{apply}(*,5,{}^{\langle\!\langle\texttt{(send a :foo)}\rangle\!\rangle}2) \\
&= {}^{\langle\!\langle\texttt{(send a :foo)}\rangle\!\rangle}\text{apply}(*,5,2) \\
&= {}^{\langle\!\langle\texttt{(send a :foo)}\rangle\!\rangle}10
\end{aligned}
$$

The final line represents an expression:

```
(progn (send a :foo) 10)
```

which properly preserves the I/O operations in the original program.

3.2.2 Handling Side-Effects (2): Instance Variable Assignment

Because ABCL/R3 is an object-oriented language, there are assignment operations to instance variables in base-level programs. In usual meta-circular interpreters, an assignment operation in the base-level program is represented as a destructive operation (e.g., `rplacd` of Lisp), which is one of the most difficult issues in partial evaluation. Rather than improving the partial evaluator to this problem, we avoid this difficulty by (1) designing the meta-level to interpret base-level assignment operations without using side-effects, and (2) reconstructing assignments *after* the partial evaluation through post-processing.

[7] This conversion is similar to a technique called *lambda-lifting.*

The meta-interpreter 'shown' to the user for reflective programming is in *direct style,* in which the environment is represented as an association list. During pre-processing, the definition is converted into *store passing style* in addition to CPS conversion, so that an assignment operation at the base-level is represented as copying of an environment list at the meta-level. The resulting evaluator functions processed by the partial evaluator takes three arguments: an expression, an environment, and a continuation, which is a function that takes two arguments: a result and an updated environment.

The problem of store passing style is that an assignment operation in the base-level program is translated into a creation of a new variable in the residual code of partial evaluation, and the original variable is not updated at all. To resolve this, we insert functions that explicitly update instance variables at the end of method execution, and then reconstruct the actual assignment (i.e., `setf`) forms during post-processing. In the compiled program, execution of assignment operations might be delayed until the end of a method, but this is not a problem for ABCL/R3 since the order of assignment operations within a method cannot be observed from other objects. For example, suppose a class `account` has a method `withdraw` defined as follows:

```
;;; class definition
(defclass account () (current 0))
;;; method definition
(defmethod account withdraw (amount)
  ;; if the request is too much,
  (if (< current amount)
    0               ; do nothing.
    (progn          ; otherwise, update the account.
      (setf current (- current amount))
      amount)))
```

When the compiler partially evaluates the default meta-interpreter `eval` with the method `withdraw`, the expression show in Figure 7 is passed on to the partial evaluator by the pre-processor.

Function `eval` interprets the assignment operation in the expression as copying of the environment value. At the end of the method, the continuation, which has a function call `update-state`, is invoked. The call to the function `update-state` is a pseudo-assignment form to the variable `current`, and will

$$
\begin{aligned}
\mathcal{PE} &: Exp \to Env \to Sval' \\
Sval' &= Preaction \times Sval \\
Sval &= Const + Pair + \cdots + Top
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{PE}[\![con]\!]\rho &= \langle\!(\,)\!\rangle\, const(con) \\
\mathcal{PE}[\![var]\!]\rho &= \langle\!(\,)\!\rangle\, \rho(var) \\
\mathcal{PE}[\![\texttt{(send } e_1\ e_2\texttt{)}]\!]\rho &= \langle\!(A_1 \cdot A_2 \cdot c)\!\rangle\, top(c) \\
&\qquad \text{where } c = \texttt{(send } s_1\ s_2\texttt{)}, \quad \langle\!(A_i)\!\rangle\, v_i = \mathcal{PE}[\![e_i]\!]\rho \\
\mathcal{PE}[\![\texttt{(+ } e_1\ e_2\texttt{)}]\!]\rho &= \text{case } (\mathcal{PE}[\![e_1]\!]\rho, \mathcal{PE}[\![e_2]\!]\rho) \text{ of} \\
&\qquad (\langle\!(A_1)\!\rangle\, const(n_1), \langle\!(A_2)\!\rangle\, const(n_2)) : \langle\!(A_1 \cdot A_2)\!\rangle\, const(n_1 + n_2) \\
&\qquad\qquad (\langle\!(A_1)\!\rangle\, s_1, \langle\!(A_2)\!\rangle\, s_2) : \langle\!(A_1 \cdot A_2)\!\rangle\, top(\texttt{(+ } s_1\ s_2\texttt{)}) \\
\mathcal{PE}[\![\texttt{(progn } e_1 \ldots e_n\texttt{)}]\!]\rho &= \langle\!(A_1 \cdots A_n)\!\rangle\, s_n \\
&\qquad \text{where } \langle\!(A_i)\!\rangle\, s_i = \mathcal{PE}[\![e_i]\!]\rho \qquad (i = 1, \ldots, n) \\
\mathcal{PE}[\![\texttt{(if } e_p\ e_c\ e_a\texttt{)}]\!]\rho &= \text{case } \mathcal{PE}[\![e_p]\!]\rho \text{ of} \\
&\qquad \langle\!(A_p)\!\rangle\, const(true) : \langle\!(A_p)\!\rangle\, \mathcal{PE}[\![e_c]\!]\rho \\
&\qquad \langle\!(A_p)\!\rangle\, const(false) : \langle\!(A_p)\!\rangle\, \mathcal{PE}[\![e_a]\!]\rho \\
&\qquad\qquad \langle\!(A_p)\!\rangle\, s_p : \langle\!(A_p)\!\rangle\, top(\texttt{(if } s_p\ \ \mathcal{PE}[\![e_c]\!]\rho\ \ \mathcal{PE}[\![e_a]\!]\rho\texttt{)})
\end{aligned}
$$

The expressions inside $\langle\!(\ )\!\rangle$ indicate the preaction of each expression.

Figure 6: Extended Partial Evaluation Rules of ABCL/R3 (abridged)

```
(eval '(if (< current amount) ; expression
           0
           (progn
             (setf current (- current amount))
             amount))
      (list (cons 'current current) ; env.
            (cons 'amount amount))
      #'(lambda (result env) ; continuation
          (update-state
            'current (lookup 'current env))
          result))
```

(Note that variables *current* and *amount* are regarded as 'unknown' by the partial evaluator.)

Figure 7: Expression to be Passed onto the Partial Evaluator

be replaced with a `setf` form via post-processing. The code yielded by the partial evaluator is shown in Figure 8.

The residual code contains function calls to `update-state` as well as obvious redundancies such as unnecessary variable references. They are also resolved via post-processing (see Section 3.3).

```
(if (< current amount)
  (progn
   (update-state 'current current)
   0)
  (progn
   (update-state 'current (- current amount))
   amount))
```

(Some redundancies, which will be removed in the post-processing phase, are already removed here for the clarity.)

Figure 8: Residual code yielded by the partial evaluator

3.2.3 Dynamic Modification of Meta-Level

Many reflective systems can alter its meta-level at run-time. This feature is useful to describe "dynamic" behavior of a system, by changing the interpretation of a base-level program according to a run-time condition. This flexibility, however, prevents the partial evaluator from eliminating interpretation because static information of the target program (i.e., the meta-interpreter) virtually diminishes.

Our solution is to apply partial evaluation as if

the meta-level were statically fixed, and support dynamic modifications to the meta-interpreter outside of the partial evaluation framework. Firstly we have restricted ABCL/R3 so that a base-level object is allowed to choose its meta-interpreter only at its creation-time[8]. Experience in reflective programming shows that this is only a slight restriction, as almost all meta-level objects determine their meta-level interpreters at its creation. For a single method of a base-level object and definitions of meta-interpreters (specified by classes of evaluator objects), *multiple compiled methods* are generated for each interpreter. On creating an object, appropriate compiled method is selected according to the specified interpreter.

Let us describe this mechanism more formally. For brevity, assume there is no delegation used among evaluator objects; i.e., the definition of an interpreter is virtually a single function. Let meta-interpreter definitions be $E_1, \ldots, E_m$, and methods of a base-level class be $M_1, \ldots, M_n$. The compiler generates a list of method tables for the class. The i'th element of the list is a method table that has compiled methods based on E_i; the j'th entry of the table has the compiled code based on $PE(E_i, M_j)$. When an object is created with a specification of the k'th meta-interpreter, the k'th method table is installed as its method table.

3.3 Post-processing

Residual programs from the partial evaluator, like the one shown in Section 3.2.2, is not itself runnable. Moreover, they may have redundancies that could be harmful to the optimizations of the back-end compiler. Residual programs are converted and optimized into ABCL/f programs in the post-processing phase, including:

Removing redundancies: Redundancies in the residual code, such as unnecessary let-bindings, unused variable references, nested `progn` forms, etc., are removed.

Reconstructing assignments: A function call to `update-state` (cf. Section 3.2.2) is converted to a `setf` form, which is an assignment form in ABCL/f.

Adding a method interface: The residual code is converted into a method definition of ABCL/f so that it has the same method interface as the original one.

The residual code shown in Section 3.2.2 is converted into the following ABCL/f method. We can observe that *all* 'interpretations' are effectively compiled away, i.e., a program identical to the original one is generated in this case.

```
(defmethod account withdraw (amount)
  (if (< current amount)
   0
   (progn (setf current (- current amount))
          amount)))
```

4 Performance Measurements

4.1 Basic Performance: Interpretation Overhead

We have performed preliminary benchmarks using a prototype ABCL/R3 compiler based on our framework. The first benchmarks compare the sequential execution speed of the the interpreter and our compiler to illustrate the effectiveness of 'compiling away' the unnecessary interpretation. Sequential benchmark programs (Boyer and n-Queens problem) are written in ABCL/R3 without using parallel constructs, nor reflective operations (although side-effects are employed). The programs are executed in three styles: (CL) compiled without the meta-level and directly executed, (INT) executed by a CPS interpreter for ABCL/R3, and (PE) the meta-level is effectively 'compiled away' using our compiler. Programs are executed on a workstation (SUN Sparcstation 10: SuperSparc 50MHz, 128MB memory) with two Common Lisp compilers (Allegro CL 4.1 and CMU CL 17e) as the back-end compiler[9].

[8] Replacement of the meta-interpreter after object's creation could be possible with more elaborate run-time support.

[9] For this benchmark, we used Common Lisp compilers, instead of the ABCL/f compiler as the back-end compiler for the following reason. The current ABCL/f compiler does not support function closures, which is necessary for execution of the interpreter in (INT). In order to do a *fair* comparison, we judged that we should employ the same back-end compiler.

	CL	PE	INT	PE	INT
Boyer	2.06	2.02	2349	0.99	1143
	1.62	1.71	269	1.058	166
8-Queens	0.043	0.050	390	1.16	9073
	0.191	0.190	34.6	0.999	182
10-Queens	1.19	1.14	9363	0.965	7901
	4.45	4.19	1011	0.940	227
	Elapsed time(sec.)			Relative speed	

Top and bottom numbers in each row correspond to the execution on Allegro CL and CMU CL, respectively. Columns in relative speed show elapsed time relative to the CL case.

Table 1: Performance Comparison between Compiled and Interpreted Executions

From Table 1, we can observe that (1) our compilation scheme exhibits equivalent performance to traditional (i.e., non-reflective) compilers, and (2) compared to naive interpretation, our compilation scheme improves performance more than 100-fold[10].

4.2 Overhead of Meta-Level Programming in Parallel Applications

The next benchmark is to measure the overhead caused by meta-level programming in parallel applications. We compare the executions in three ways. (Original) The original program without meta-level optimizations is directly compiled by the ABCL/f compiler, and executed on Fujitsu AP1000, a massively parallel processors with 64 Sparc-based nodes and very fast torus network interconnection[22]. (Hand-craft) The application is *manually* optimized (see below) and compiled by the ABCL/f compiler. (Meta) The same optimizations are extracted and separately specified as a meta-level class library, and the original program at the base-level is not modified except for a few annotations; these programs are compiled together by our compiler, and executed.

Target application programs are as follows:

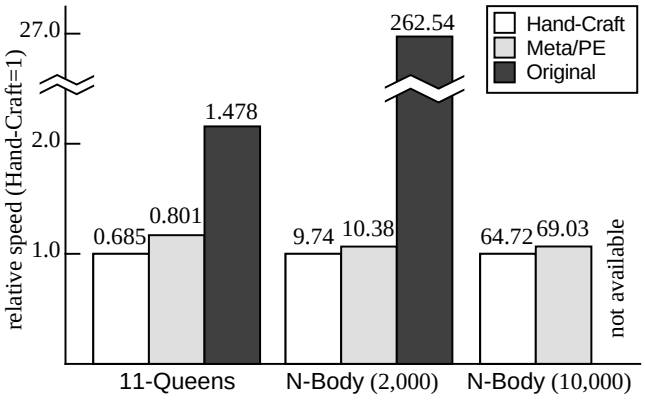
Parallel Search: The first base-level application is a simple parallel search program (n-Queens problem). Each object is generated as a node in the search tree. Optimizations in Hand-craft and Meta are: (1) *Locality control*—child nodes (objects) at deep levels in the search tree are created at the same processor as their parents' in order to reduce remote communication overhead (the default is to randomly choose a processor). (2) *Weighted termination detection*[19]—'weight' is propagated along the search tree in order to detect the end of a search process. By default, the detection is achieved by collecting acknowledgments in the search tree; therefore, intermediate search nodes cannot be released until all its descendant nodes terminate. The meta-level program and its compiled code in the Meta case are given in Appendix A.

N-Body Simulation The second base-level application is a parallel Barnes-Hut N-body simulation algorithm. The optimization technique employed in a hand-tuned ABCL/f code is to cache sub-space data, and exhibits comparable performance to highly optimized algorithm presented in [8]. In Hand-craft, method calls that access subspaces in the base-level program are modified to first look-up the cache. In ABCL/R3, this optimization is separately described at the meta-level; a customized meta-interpreter is defined that looks up the cache on specific method calls.

The graph in Figure 9 shows the benchmark results of above two applications: 11-Queens problem, and 2,000/10,000 particles N-body simulations. All programs are executed on Fujitsu AP1000 (64 $\times$ (25MHz Sparc processor + 16MB memory)). From the graph, we can observe that the Meta execution (1) significantly improves the performance of the

Fortunately the sequential part of ABCL/R3 is almost identical to Common Lisp; thus, we can easily convert sequential ABCL/R3 program into Common Lisp programs by replacing message sends with function calls, for example. Note that this was done for benchmark purposes only; since under normal circumstances the partial evaluator unfolds possible function applications, the residual code, compiled with the ABCL/f compiler, does not contain function closures.

[10] The interpreter used in this benchmark is not highly optimized. However, it is worth pointing out that previous studies to optimize/minimize interpreters still result in a factor of 10 times slower execution compared to the non-reflective compilers even with limited 'openness'[6, 14].



The height of each bar shows elapsed time relative to Hand-craft. Figures on top of each bar are real elapsed time in seconds. The Original execution of N-Body (10,000) failed because of memory exhaustion.

Figure 9: Comparison of Overhead of Meta-Level Programming

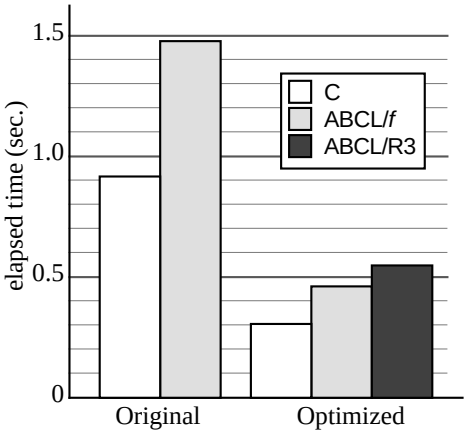


Figure 10: Elapsed time for 11-Queens problem in C, ABCL/f, and ABCL/R3

Original program, and (2) has only small overhead compared to the Hand-craft one, while encapsulating the optimizations into the meta-level. (In the n-Queens problem, the overhead was about 17%. In the N-body simulation, the overhead in both cases was approximately 7%.) Consequently, we have achieved high efficiency as well as good programmability and re-usability at the same time.

The source of the overhead is mainly that (1) the partial evaluator converts a loop in the base-level program into recursive functions, which is less efficient in ABCL/f, (2) management of 'weights' for termination detection is implemented as separate methods, while they are inlined into the search function in the Hand-craft case, (3) unnecessary assignments of instance variables are performed because of the technique described in Section 3.2.2. The overhead could be reduced by doing further optimizations such as eager inlining as in Self[4], and static flow analysis.

To investigates the baseline efficiency of above programs, we also executed benchmark programs written in C with a message-passing library, against those written in ABCL/f and ABCL/R3 on the AP1000. The left bars in Figure 10 indicate elapsed times for the execution of 11-Queens problem (Original) written in C and ABCL/f. The right bars are optimized ones in C, ABCL/f, and ABCL/R3. Only the *locality control* technique is employed here; it is achieved by modifying the base-level application (C and ABCL/f), or customizing the meta-level (ABCL/R3). We observe that (1) ABCL/f is only 1.5–1.6 times slower than C, and (2) the optimization effectively improves performance about by 3-fold both in ABCL/f and C.

5 Discussions and Related Work

Optimization techniques in ABCL/R2[14] assumes that (1) a predetermined set of operations is interpreted and can be modified by reflective programming, whereas other operations are non-reflective and thus cannot be modified so as to preserve their efficiency, and (2) some system-defined objects are subject to *lazy reification*, so that they could be executed directly until the system detects that interpretation is required. Similar approaches are believed to be taken by CLOS-MOP and AL-1/D[17]. Although a dramatic improvement over pure interpretation, our experience is that even a small number of interpreted operations drastically decrease performance. Even in Open C++[5], which throws away the ideal of presenting the entire meta-circular interpreter, but rather only provides a restricted set

of customizable operations (namely method invocation, object creation, and instance variables access), any customization incurs a form of interpretation overhead.

Some researchers investigated the alternative of subsuming the interpretation overhead with algorithmic gains achieved by reflective programming. For example, Okamura, et al. illustrate the effectiveness of meta-level optimization in a distributed information retrieval system[16]. Their underlying computing environment exhibits very low remote message passing performance, and benefit of reducing the number of remote messages overcomes the overhead of reflection. However, since such assumptions are not universal, one cannot always make such a tradeoff; for example, with massively parallel machines, message passing latency is a factor of 100–1000 times faster compared to workstation TCP/IP communication over Ethernet, so gains in communication would be overwhelmed by interpretation overhead. Furthermore, forcing the users to consider the tradeoff between interpretation vs. performance improvement complicates the applicability of open implementation technology. For example, if a particular algorithm change at the meta-level does not improve system performance, it will be difficult to judge whether the reason was the algorithm itself, or the interpretation overhead.

Instead, in our approach, we can present the programmer with customizability of the entire meta-interpreter described concisely in object-oriented style. The compiler successfully eliminates much of the unnecessary run-time interpretation overhead. The programmer no longer needs to be concerned with tradeoffs incurred with interpretation, widening the applicability of open implementation technology to areas including time-critical and performance conscious ones. Although there was an earlier static optimization attempt of meta-level code in Open C++[6] with a similar objective, its main optimization was in elimination of idempotent reify/reflect pairs, and did not have the full generality or efficiency of our compiler.

One might also claim that meta-level optimization and customization examples we have presented might well have been done solely at the base-level. However, for large-scale programming, meta-level programming can describe such changes in a concise, modular manner, and ubiquitously apply it throughout the program (with of course, appropriate scope control). For example, it would be unfeasible to expect the programmer to chase through a 100,000-line program to insert notification code every time there is a syntactic reference to a variable `client`, and then change it back when notification is no longer required. Being able to describe the changes in a modular, object-oriented fashion is the essential characteristics for creating customizable meta-level class frameworks.

As we have stated earlier, 'compiling away' meta-level interpretation requires the full generality of semantic-based optimization through partial evaluation, and cannot be achieved by standard optimization techniques such as inlining. This is not to say that standard optimization techniques are useless; rather, there are numerous possible local optimizations that the partial evaluator does *not* support, only a few of which the current compiler performs during post-processing. More advanced optimization techniques developed in high-performance object-oriented compilers such as Self[4] and Concert[11] could be greatly beneficial in this regard. Moreover, partial evaluator enlarges the opportunity for advanced optimizations because the residual code is directly executable without any indirections caused by interpretation.

Since the discovery of the Futamura projection[7], there have been a number of studies that uses partial evaluation to compile programs from interpreter definitions[9, 21]. The use of partial evaluation in our work follows the same path, albeit special techniques employed so as to be applicable to concurrent object-oriented programming. Partial evaluation is not a panacea; however, we believe that other program transformation techniques as well as run-time techniques are also important in order to build realistic reflective systems.

6 Conclusion

In this paper, we proposed a compilation framework for object-oriented concurrent reflective lan-

guages based on partial evaluation that almost completely 'compiles away' the overhead of meta-level interpretation. The techniques that make partial evaluation possible include (1) pre-processing that converts object-oriented evaluator definition to CPS (and store-passing) functions, (2) a new partial evaluation technique called *preaction* that preserve I/O type side-effects, (3) multiple compiled methods to cope with dynamic modification of the meta-level, and (4) post-processing that resolves assignments to instance variables and performs some other optimizations.

Preliminary benchmarks indicate that (1) the sequential reflective programs in our framework exhibit equivalent performance to the ones compiled by non-reflective compilers, (2) compiled programs are faster than interpreted ones by orders of magnitude, (3) optimizations that are separately described by a meta-level class framework applied to a parallel application poses only 10–30% overhead, compared to a program that has been hand-tuned by embedding optimizations, and compiled by a non-reflective compiler.

Acknowledgments We would like to thank Kenjiro Taura for the his assistance in using ABCL/*f* system, and many suggestions on our work. We also thank Shigeru Chiba, Yuuji Ichisugi, and Jean-Pierre Briot for their helpful comments.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[2] K. Asai, 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, 1995. to appear.

[3] R. Baier, R. Glük, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In *PEPM'94*, June 1994. Published as Technical Report 94/9, Department of Computer Science, University of Melbourne.

[4] C. Chambers and D. Ungar. Making pure object-oriented languages practical. In *OOPSLA'91*, pp. 1–15, Oct. 1991.

[5] S. Chiba and T. Masuda. Designing an extensible distributed language with a meta-level architecture. In *ECOOP'93*, 1993.

[6] S. Chiba and T. Masuda. Open C++ and its optimization. In *Proceedings of OOPSLA'93 Workshop on Object-Oriented Reflection and Metalevel Architectures*, Sept. 1993.

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

[8] A. Y. Grama, V. Kumar, and A. Sameh. Scalable parallel formulation of the Barnes-Hut method for n-body simulations. In *Supercomputing'94*, pp. 439–448, 1994.

[9] A. Haraldsson. A partial evaluator, and its use for compiling iterative statements in Lisp. In *POPL'78*, pp. 195–202, 1978.

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

[11] V. Karamcheti and A. Chien. *Concert*—efficient runtime support for concurrent object-oriented programming languages on stock hardware. In *Supercomputing'93*, pp. 598–607, Nov. 1993.

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

[13] J. Lamping, G. Kiczales, L. Rodriguez, and E. Ruf. An architecture for an open compiler. In *Proceedings of IMSA workshop on Reflection and Meta-Level Architecture*, pp. 95–106, Nov. 1992.

[14] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *OOPSLA'92*, pp. 127–145, Oct. 1992.

[15] U. Meyer. Techniques for partial evaluation of imperative languages. In *PEPM'91*, pp. 94–105, June 1991.

[16] H. Okamura and Y. Ishikawa. Object location control using meta-level programming. In *ECOOP'94 (LNCS 821)*, pp. 299–319. July 1994.

[17] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/D: Distributed programming system with multi-model reflection framework. In *Proceedings of IMSA workshop on Reflection and Meta-Level Architecture*, pp. 36–47, Nov. 1992.

[18] L. Rodriguez, Jr. A study on the viability of a production-quality metaobject protocol-based statically parallelizing compiler. In *Proceedings of IMSA workshop on Reflection and Meta-Level Architecture*, pp. 107–112, Nov. 1992.

[19] K. Rokusawa, N. Ichiyoshi, T. Chikayama, and H. Nakashima. An efficient termination detection and abortion algorithm for distributed processing systems. In *International Conference on Parallel Processing*, vol. I, pp. 18–22, 1988. also published as ICOT-TR 341.

[20] S. A. Romanenko. Arity raiser and its use in program specialization. In *ESOP'90 3rd European Symposium on Programming (LNCS 432)*, pp. 341–360, May 1990.

[21] S. Safra and E. Shapiro. Meta interpreters for real. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, vol. 2, chapter 25, pp. 166–179. MIT Press, 1987.

[22] T. Shimizu, T. Horie, and H. Ishihata. Low-latency message communication support for the AP1000. In *International Symposium on Computer Architecture*, vol. 20, pp. 288–297, May 1992.

[23] K. Taura, S. Matsuoka, and A. Yonezawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language – its design and implementation –. In *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, 1994.

[24] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *OOPSLA'88*, pages 306–315, Sept. 1988.

[25] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In *Functional Programming Languages and Computer Architecture (LNCS 523)*, pp. 165–191. Aug. 1991.

A Example Compilation of Reflective Programs in ABCL/R3

Here we give a meta-level programming example—the n-Queens problem described in Section 4.2, and the excerpt of the actual compiled result.

The Base-level Program The base-level program is an n-Queens parallel search problem. A search node in the search tree is represented by a concurrent object.

```
(defclass queen ())
(defmethod queen do-search (size col places)
  (if (= size col) ; do we have an answer?
    ;; yes, report the answer
    (print-answer *printer* places)
    (dotimes (i size)
      ;; no, check if we can place at i'th row of
      ;; the next column
      (when (not (checked? col i places))
        ;; create a new object and
        ;; have it search in parallel
        (past (do-search (new 'queen) size
                (1+ col) (cons i places)))))))
```

The Evaluator for Locality Control The meta-level programs are divided into two modules; the locality control module and the weighted termination detection module. Firstly, locality control is achieved by the evaluator object `locality-eval`, which specifies the processor numbers of newly created objects. A meta-level argument `depth` is transparently added to inter-object message passing.

```
;;; Class definition; inherits from class standard-eval
(defclass locality-eval (standard-eval))

;;; The method that gives the processor number for
;;; object creation is overridden.
(defmethod locality-eval
  get-object-creation-node (class arg-vals env)
  ;; look up the meta-level variable depth
  (let ((depth (lookup-meta-var 'depth env)))
    ;; compare with the threshold
    (if (< *threshold* depth)
      ;; create on the local processor
      (this-node-id)
      ;; create on a remote processor
      (random-node-id))))

;;; A hidden parameter depth is passed to a newly
;;; created object. The following method returns an
;;; association list of parameter names and values.
(defmethod locality-eval
             get-object-creation-meta-arg
               (class arg-vals env)
  ;; look up the meta-level variable depth
  (let ((depth (lookup-meta-var 'depth env)))
    (cons ;; (current depth)+1
      (cons :depth (1+ depth))
      ;; combine parameter list with delegatee's
      (delegate))))
```

The Evaluator for Weighted Termination Detection The module for the weighted termination detection modules manager objects, evaluator object `WTD-eval`, and several meta-object methods. Here, we only show the evaluator, which (1) calls an initialization method at the beginning of the

```
(defmethod queen do-search (size col places)
  (meta-init-weight self)                              ; from WTD-eval
  (if (= size col)
    (progn (print-answer *printer* places)
           (meta-return-weight self))                  ; from WTD-eval
    (if (< 0 size)                                     ; **
      (if (not (checked? col 0 places))
        (if (< *threshold* depth)                      ; from locality-eval
          (progn
            (past (do-search
                    (new 'queen :depth  (1+ depth)
                                :weight (meta-weight-for-child self)
                                :on (this-node-id))    ; local creation
                    size (1+ col) (cons 0 places)))
            (eval-while818 depth col size 1 places))   ; next step of the loop
          (progn
            (past (do-search
                    (new 'queen :depth (1+ depth)
                                :weight (meta-weight-for-child self)
                                :on (random-node-id)) ; random creation
                    size (1+ col) (cons 0 places)))
            (eval-while819 depth col size 1 places))) ; next step of the loop
        (eval-while820 depth col size 1 places))       ; next step of the loop
      (meta-return-weight self))))                     ; from WTD-eval

;;; Methods eval-819, 820 have same definition.
(defmethod queen eval-while818 (depth col size row places)
  ;; The body is almost identical to the lines after '**' of the method
  ;; do-search except that the value 0 is replaced with the variable row.
  )
```

Figure 11: Compiled Result of n-Queens Program

base-level method, (2) distributes weight to child objects, and (3) calls a finalization method (to return weight) at the termination of the base-level method.

```
;;; Class definition.
(defclass WTD-eval (standard-eval))

;;; Invoke method init-weight at the beginning of a
;;; method.
(defmethod WTD-eval eval-entry-method (exp env)
  ;; variable ID refers the meta-object
  (init-weight ID)
  ;; body of method execution (by delegation)
  (delegate))

;;; Parameter weight is passed on to child objects.
(defmethod WTD-eval
            get-object-creation-meta-arg
              (class arg-vals env)
  (cons (cons :weight
              (get-weight-for-child ID))
        (delegate)))

;;; Invoke the method to return weight at termination
;;; of a method.
(defmethod WTD-eval eval-exit-method
                                  (return-value env)
  (return-remaining-weight ID) ; return weight
  (delegate))
```

Compiled Code We show the resulting compiled code in Figure 11 before being passed into the back-end compiler. Some arguments have been omitted, and some variables have been renamed for readability. Although program size has become slightly larger, interpretation is 'compiled away.' The reasons for increase in program size are: (1) a loop in the original program has been converted to recursive functions, (2) code after a conditional expression has been duplicated, (3) the first iteration of the loop has been unfolded, and (4) different specialized function is constructed for each branch of conditionals, although they have the same definitions.