to appear in Proceedings of OOPSLA'98 workshop on Reflective Programming in C++ and Java

On-the-fly Specialization of Reflective Programs Using Dynamic Code Generation Techniques*

SUGITA, Yuuya[†]
HARADA,Ken'ichi[†]
MASUHARA, Hidehiko[‡]
YONEZAWA, Akinori[§]

1 Introduction

In these days, many distributed application programs require *dynamic adaptation* to their execution environment as the environment dynamically evolves, and the programs may be dynamically transported over network. Reflection could be beneficial to dynamic adaptation in its nature. A running base-level program can adapt to an environment by replacing its meta-program with a new one which embodies an environment-specific policy. For example, Amano and Watanabe proposed a reflective architecture suitable for dynamic adaptation[1].

Unfortunately, efficient implementation techniques of reflective languages, such as the *partial evaluation*[2, 9, 10] and the *compile-time reflection*[4, 7], do not suit dynamic adaptation because the programs should be fixed for compilation. In other words, when a program is 'migrated' to a different meta-program, a sluggish compilation process must be performed dynamically.

This paper proposes a novel way to implement reflective languages using the *dynamic code generation* (DCG) techniques[3, 5, 8]. As far as the authors know, this is the first study that uses DCG for dynamically specializing reflective programs. Advantages of DCG-based implementation are that: (1) the specialized process is faster than the partial evaluation/compilation process by orders of magnitude; and (2) the interpretation overheads in the meta-interpreters in the specialized program, as well as the partial evaluation based implementation.

However, for successful specialization, the DCG techniques pose more restrictions on the target programs. Especially, the structure of meta-interpreters seems to cause performance problems in DCG.

In order to clarify the problems, we first introduce the basic mechanism of a DCG system (Sec. 2). We then define a Scheme-based simple reflective language (Sec. 3), and measure basic performance of DCG specialized meta-interpreters (Sec. 4). As expected, they are not fast enough. We thus examine the reasons of overheads and ideas to reduce them (Sec. 5).

Although we use a simple reflective language, we believe that the discussion in this paper would also be applicable to practical reflective languages such as C++ and Java based ones.

2 Dynamic Code Generation

2.1 Specialization by Using Partial Evaluators

Several studies have shown that reflective programs can be compiled by using partial evaluation[2, 9]. Assume that *PE*, *int*, and *p* are a partial evaluator, a meta-interpreter, and a base-level program, respectively. The compilation process, also known as the first Futamura projection[6] is:

*This study was supported in part by the Japanese Ministry of Education Grand-in-Aid for Scientific Research on Priority Area: "Construction Principles for Software with Evolution Mechanisms."

[†]*Dept. of Computer Science, KEIO Univ.*
{yuuya,harada}@hara.cs.keio.ac.jp
[‡]*Dept. of Graphics and Computer Science, Graduate School of Arts and Science, Univ. of Tokyo*
masuhara@graco.c.u-tokyo.ac.jp
[§]*Dept. of Information Science, Univ. of Tokyo*
yonezawa@is.s.u-tokyo.ac.jp

$$PE(int, p) = int_p \quad (1)$$

where int_p is a result of partial evaluation; i.e., a specialized version of int with respect to p. Since partial evaluators perform source-to-source translation, int_p must be compiled into executable code, int_p^M (we let superscript M denote executable machine instructions), which would be as efficient as the code a compiler for the language defined by int, if exists, generated from p.

The problem of this scheme is that the system should perform the process (1) and subsequent compilation, whenever either p or int is modified.

2.2 Specialization by Using DCG

DCG is, similar to partial evaluation, a methodology for specializing programs. Unlike partial evaluation, however, DCG performs specialization much faster. The DCG first generates compiled code fragments for dynamic part of int, and a program for static part of int, which we call code generator $intgen^M$:

$$DCG(int) = intgen^M \quad (2)$$

When the code generator is applied to a part of int's input, say p, it generates a specialized version of int:

$$intgen^M(p) = int_p^M \quad (3)$$

This process is very efficient because it merely places the code fragments that are generated in (2). From the viewpoint of adaptation, when p is moved to a new interpreter int', the system only have to perform the process (3) with the code generator $intgen'^M$ for int', which is faster than the process (1).

2.3 Overview of DCG

Let us see how DCG works, by using a simple program as an example. A DCG takes a program in which each expression is annotated with its binding-time.

For example, a program that computes `n`'th power of `x` is annotated as follows:

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

The non-underlined expressions are static; they will appear in code generators (e.g., $intgen^M$). The underlined expressions, on the other hand, are dynamic; they will appear in the generated code (e.g., int_p^M). An `@` operator shows the binding-time of a function application: `@` means that the application is performed at code generation time; $\overline{\texttt{@}}$ means that the corresponding code generator is invoked at code generation time; and `@` means that a code fragment for the application is generated at compile time.

The templates and the code generator for this program are shown in Fig.1[1]. A template is a sequence of machine instructions that corresponds to the underlined expressions. A code generator is a function that performs the non-underlined expressions, and loads templates into the memory. When `pow-gen` is called with argument 3, it generates the instructions as shown in Fig.2, which essentially has the same functionality as the function `(lambda (x) (* x (* x (* x 1))))`.

Unfortunately, DCG generated code is not as efficient as the code generated from a statically partially evaluated result. Generally, when a code generator is called upon an $\overline{\texttt{@}}$ operator, instructions that save/restore registers, pass arguments, and adjust the frame pointer are generated. However, they may be unnecessary in the generated code. In the example in Fig.1, the template `t0` adjusts the frame pointer, and the templates `t2` and `t3`, passes an argument through the stack.

```
push %ebp; movl %esp,%ebp; push -8(%ebp);
 push %ebp; movl %esp,%ebp; push -8(%ebp);
  push %ebp; movl %esp,%ebp; push -8(%ebp);
   movl $1,%eax; leave;
  addl $4,%esp; imull -8(%ebp),%eax; leave;
 addl $4,%esp; imull -8(%ebp),%eax; leave;
addl $4,%esp; imull -8(%ebp),%eax; leave;
```

Figure 2: DCG generated instructions.

3 A Simple Reflective Language

3.1 Reflective Architecture

For simplicity, our reflective architecture has only the base-level and the meta-level. Both levels are functional subset of Scheme. At the meta-level, the user can define customized meta-interpreters, each of which is a function that takes an expression, an environment and a store, and returns a

[1] In the figure, templates are written in i386's mnemonic. `%eax` is a general register and `%esp` is a stack pointer. A function receives its arguments in the stack, and returns a result in `%eax`.

Templates

```
t0 :  push %ebp     /* function preamble */
      movl %esp, %ebp
t1 :  movl $1, $eax          /* return 1 */
      leave          /* function trailer */
t2 :  push -8(%ebp) /* pass argument */
t3 :  addl $4, %esp
      imull -8(%ebp),%eax /* x*pow(x-1) */
      leave          /* function trailer */
```

Dynamic Code Generator

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

Figure 1: Templates and code generator for `pow`.

value for the expression. They are treated as first-class objects in base-level programs. The base-level language has a reflective operation `get-eval` to obtain a reference to a meta-interpreter. When a form `(eval-with` e_1 e_2`)` is evaluated at the base-level, the expression e_1 and e_2 are first evaluated under the current meta-interpreter, and then the result of e_1, which should be an S-expression, is evaluated under the result of e_2, which should be a reference to a meta-interpreter.

3.2 Implementation of `eval-with`

We use a DCG technique to implement the `eval-with` form. At the compile time, the system generates a code generator for each meta-interpreter. In Fig.3, two generators *int-a-gen* and *int-b-gen* are generated for the customized meta-interpreters *int-a* and *int-b*, respectively. When the form `(eval-with` e_1 e_2`)` is evaluated, e_1 and e_2 are first evaluated to p and *int-a*, the the run-time system looks up the specialized code cache with a key $\langle p$,*int-a*$\rangle$. If the cached code is found, the system merely executes it. Otherwise, it applies *int-a-gen* to p, yielding the specialized code *int-*a_p^M. The system puts the generated code in the cache with a key $\langle p$,*int-a*$\rangle$, and executes it.

4 Preliminary Performance Evaluation

In order to evaluate the basic performance of our specialization framework, we have constructed a DCG system for Scheme subset, which is powerful enough to manipulate function closures, structured data, etc., and in fact it successfully specialize a Scheme interpreter in Scheme.

Tab.1 shows the execution times of benchmark programs that are interpreted by a default meta-interpreter, and are compiled by four specialization techniques; all use DEC Scheme-to-C compiler for generating native instructions. The programs are executed on a Pentium box running at 150MHz with 48MB memory.

Table 1: Execution times [*sec*].

	matrix	deriv
no specialization	2.25	4.03
dynamic specialization	0.481	0.770
static specialization	0.400	0.602
ideal	0.023	0.051

The matrix benchmark performs 100 multiplications of 5×5 matrices. The deriv benchmark symbolically computes differentiations of functions for 500 times. The DCG specialized programs, whose execution times are displayed in the **dynamic specialization** row in Tab.1, improve the performance over the interpreter-based programs, displayed in the **no specialization** row, by more than the factor of 4. Tab.2 shows the times spend in specializing the benchmark programs by using DCG and the partial evaluator. The DCG improves the specialization times by three orders of magnitude. We thus conclude that our framework can be used as performance improvement from the interpreter-based implementations for most cases. The DCG specialized programs are 20–30 percent slower than the programs statically partially evaluated by Similix, a Scheme partial evaluator, whose execution times are displayed in the **static specialization** row. The causes and planned remedies of this overhead are discussed in the next section. However, as the specialization times of those techniques show, this overhead could be covered for dynamically adapted programs with the faster adaptation process. Our current meta-interpreters are not sufficiently specialized in both dynamic and static

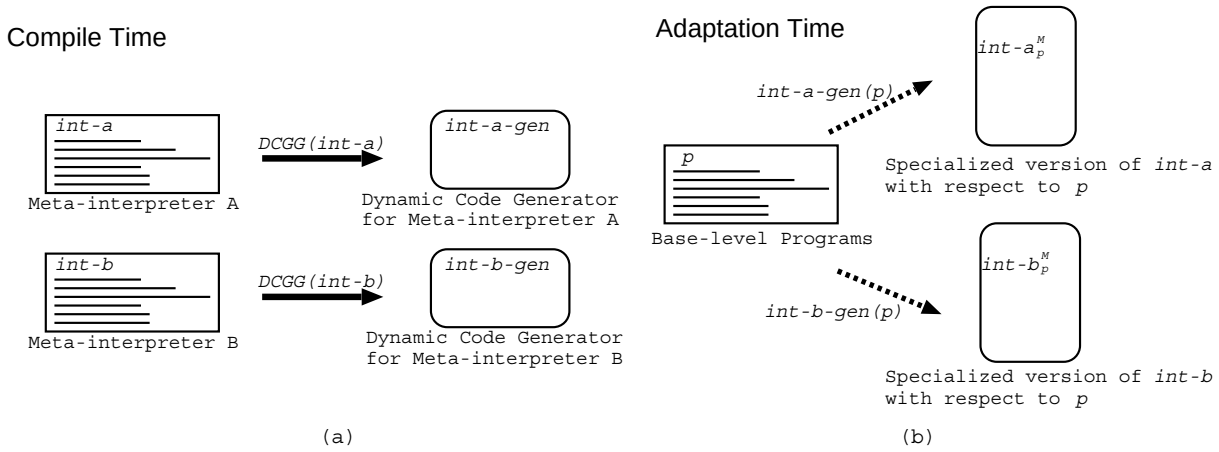

Figure 3: Outline of implementation.

Table 2: Specialization times[*sec*].

	matrix	deriv
dynamic specialization	0.005	0.006
static specialization	10.5	14.7

techniques. The **ideal**[2] row displays the execution times of the benchmarks that are directly compiled without having the meta-interpreter. Those programs are more than 11-fold faster over the specialized programs, which are specialized either statically or dynamically. We conjecture that the current convention for base-level parameter passing causes inefficient variable references in the specialized programs, and that the representation of base-level closures interferes the Scheme compiler's optimization techniques. Some of those overheads could be eliminated by cleverly designed meta-interpreters. We discuss these issues in the next section.

5 Planned Techniques to Improve Efficiency

As mentioned above, the DCG specialized programs still have a considerable amount of overheads. Below, we discuss the reasons of those overheads, and the techniques that are planned to reduce them.

[2]We call this the ideal case since an ideal specializer would return the original base-program for a default meta-interpreter.

5.1 Inlining to Remove Unnecessary Register Saving Operations

As we have seen in Sec.2.3, when an input program has `@` operators, the specialization result of DCG could contain unnecessary instructions such as the ones for saving registers, which degrades performance of generated code. A meta-interpreter may suffer this performance degradation due to its peculiar programming style. It recursively defines the semantics of the base-language *for each programming constructs.* This means that even for a light expression, such as a variable reference and a constant, its 'compiled' code fragment include the unnecessary instructions. For example, when the code generator for `eval` in Fig.4 is applied to the expression `(eq? 1 x)`, unnecessary instructions will be generated around `1` and `x` because the definition for `eq?` has two `@` applications.

To eliminate those unnecessary instructions, we plan to perform preprocessing that inlines recursive calls of `eval` before the compilation, so as to decrease the number of `@` applications. For the time being, we are examining appropriate ways to do so, but of course, inlining techniques could easily result in code explosion. We are therefore investigating techniques to inline programs for specific base-expression patterns.

```
(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 4: The definition of `eq?` form.

5.2 Stack Manipulation Primitives for Efficient Parameter Passing

The parameter passing for base-level functions in our meta-interpreter uses list structure to cope with an arbitrary number of arguments. Since a specialized program basically inherits data structure from its original program, the specialized programs also have to perform inefficient list manipulations for function calls. The *arity raising*[11], which is known to be useful to this problem, is inappropriate for dynamic adaptation because it relies on a global analysis of a specialized program.

Our plan is to provide primitives that push/pop arguments for base-level functions, and to define meta-interpreters so that they use those primitives for base-level function calls. At compilation-time, by translating those primitives into the instructions that manipulates the stack, registers, or register window, the parameter passing in the specialized programs would be performed in more efficient manner.

6 Conclusion

In this paper, we have proposed an on-the-fly specialization framework of reflective programs. At compile-time, the system generates a dedicated code generator for each meta-interpreter by using a DCG system, and, at the run-time, the code generator yields a specialized executable program for given base-level programs. Our prototype system showed improvement of execution times of benchmark programs by more than 5-fold over interpreter-based execution, and improvement of specialization times by orders of magnitude over the specializations using partial evaluators.

The benchmarks also showed that our meta-interpreters are not yet sufficiently specialized. We examined some causes of the problems and proposed techniques to remedy the overhead, namely an inlining technique for enlarging template granularity, and efficient parameter passing primitives for base-level functions. Those will be further investigated.

Acknowledgments

The authors would like to thank Satoshi Matsuoka, Ken'ichi Asai, and Kenjiro Taura for their valuable advises and discussions.

References

[1] N. Amano and T. Watanabe. A procedual model of dynamic adaptability and its description language. In *International Workshop on Principles of Software Evolution,* pp. 103–107, April 1998.

[2] K. Asai, S. Matsuoka and A. Yonezawa. Duplication and Partial Evaluation —For a Better Understanding of Reflective Languages—, In *Lisp and Symbolic Computation,* Vol. 9, pp.203-241, 1996.

[3] C. Consel and F. Noël. A general approach for run-time code generation and its application to C. In *PoPL'96*, pp. 145–156, January 1996.

[4] S. Chiba. A metaobject protocol for C++. In *OOPSLA'95*, pp. 285–299, 1995.

[5] D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *PLDI'96*, pp. 160–170, May 1996.

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

[7] Y. Ishikawa, et al. Design and implementation of metalevel architecture in C++: MPC++ approach. In *Reflection '96*, April 1996.

[8] P. Lee and M. Leone. Optimizing ML with run-time code generation. In *PLDI'96*, pp. 137–148, May 1996.

[9] H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In *OOPSLA'95*, pp. 300–315, 1995.

[10] H. Masuhara and A. Yonezawa. Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Language. In *ECOOP'98*, pp. 418–439, 1998.

[11] S. Romanenko. Arity Raiser and its Use in Program Specialization. In *ESOP'90*, pp. 341–360, May 1990.

[12] U. Hölzle and D. Ungar A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance In *OOPSLA'94*, pp. 229–243, 1994.