Partial Evaluation in Reflective System Implementations

Erik Ruf
Microsoft Research Laboratory
One Microsoft Way
Redmond, WA 98052-6399
ruf@research.microsoft.com

Abstract

This position paper explores the use of partial evaluation as an implementation mechanism for both compile-time and run-time metalevel architectures. Specifically, we show how the specialization of subprograms (at both the base and meta levels) can improve the performance of programs written under "open" language implementations. We also speculate on the usefulness of a meta-level interface to the partial evaluator itself.

1. Introduction

Partial evaluation (also known as program specialization) is a powerful program optimization technique. A partial evaluator transforms a program and a partial description of its inputs into a new, specialized program which, when applied to any inputs satisfying the description, computes the same result as the original program. By restricting the applicability of the specialized program, the specializer can optimize away computations that depend only on the known portion of the inputs.

Traditional uses of partial evaluation have focused on the specialization of entire programs, where the program is executed repeatedly, and where one or more of its input values remains constant while another one varies. For example, if a circuit simulator is run repeatedly on different initial conditions for a single circuit description, specializing the simulator on the circuit description can provide large benefits. However, partial evaluation can be just as powerful when applied to subprograms, in that we can construct multiple optimized versions, or variants, of a subprogram, and use them instead of the subprogram in contexts in which they are applicable. This "multiple specialization" behavior is typically called polyvariance, and has yielded significant benefits in non reflective language implementations.

In this paper, we examine the usefulness of partial evaluation in improving the runtime performance of meta-protocol-based programming language implementations. Our goal is to identify opportunities for improvement that are due, in particular, to the presence of a meta-level, as distinguished from optimizations that could be performed under an ordinary language architecture.

The remainder of this paper has 5 sections. Section 2 describes the use of dynamic specialization in run-time meta-level architectures, while Section 3 describes the use of static specialization at compile time. In section 4, we treat a difficulty in the partial evaluation of multi-level languages, and argue that a meta-protocol-based partial evaluator may be the answer. We conclude with a brief survey of related work and future directions.

2. Runtime Meta-code

Runtime meta-level architectures introduce additional overhead over base-level architectures because, to execute a base-level operation, the system must execute the meta-level code that implements it. The means for such meta-execution vary (e.g., explicit interpretation, the association of meta-objects and operations with individual base-level objects, or meta-protocols for executing language constructs) but the overhead is always there.

Typically, though, not all of the meta-code associated with a base-level operation need be executed every time the base-level operation is performed. Often, many of the decisions about how to implement a particular base-level operation depend only on parameters that change very slowly, relative to the rate of invocation of the operation. For example, in one of the example meta-object protocols described in [6], the meta-code implementing a particular generic function is parameterized not only with respect to the parameters passed to the generic function, but also w.r.t. the set of methods defined on the generic function. Since the latter changes far more slowly than the former, we could improve efficiency by specializing the meta-code w.r.t. a set of method definitions, and using the specialized meta-code until a method is added, deleted, or altered. If we can prove (or are explicitly informed) that no such modifications will occur, then all of the meta-operations depending solely on method information can be executed once, at compile time. Similar observations can be made about other base-level language constructs; e.g., the

In the original meta-code, before staging, each application of a base-level generic function invokes the meta-function apply-generic-function.

```
(define (apply-gf methods arguments)
  ;; compute the applicable methods
  ;; loop through them, applying them
  ;; to the arguments
)

;; implement two base-level gf calls
(apply-gf (methods gf) args1)
(apply-gf (methods gf) args2)
```

After staging, we compute a specialized application function once, then invoke that on each application of the generic function:

```
(define (compute-gf-applier methods)
  ;; compute the applicable methods
  (lambda (arguments)
    ;; loop through them, applying them
    ;; to the arguments
  ))

;; implement two base-level gf calls
(define app (compute-gf-applier
              (methods gf))
(app args1)
(app args2)
```

Figure 1. Simplified example of staging in runtime meta-code implementing generic functions.

storage allocation for an a slot in an object (e.g. an instance variable) will change less frequently than the slot is accessed, suggesting that we compute specialized slot access code.

This observation is not new. Indeed, at least one existing implementation, the CLOS Meta-Object Protocol [6] relies on it heavily. Given the knowledge that method definitions change infrequently relative to generic function invocations, the CLOS implementors manually reordered the meta-code such that computations dependent solely on slowly-changing values are performed once, and are used to produce specialized meta-code that is parameterized solely on the rapidly-changing values.[1] A simplified version of this transformation is shown in Figure 1.

We use the term "staging" to denote this division of operations into early (performed rarely, when specialized code is constructed) and late (performed frequently, each time the specialized code runs) categories. In the case of CLOS, further partial evaluation of the code gives little benefit. Indeed, humans may be better than programs at performing such divisions, as they have knowledge of invariants that partial evaluators cannot discover themselves.

What partial evaluation *does* offer is a means to at least partially automate such staging. The knowledge of the relative "volatility" of parameters to the meta-code would still need to be provided by the implementor, but, given that, a partial evaluator could automatically produce the re-staged meta-code. The code for `compute-gf-applier` in Figure 1 would become something like:

```
(define (compute-gf-applier methods)
  (compile
    ;; partially evaluate apply-gf
    ;; with respect to known methods
    ;; but unknown arguments
    (pe apply-gf methods *unknown*)))
```

The code produced by the partial evaluator would be similar to that of Figure 1 (and that of p. 127 in [6]), except that, since the actual methods are available at partial evaluation time, much of the looping and application code in the body of apply-gf would also be inlined and optimized.[2] However, there are two possible performance pitfalls. First, the partial evaluator must analyze apply-gf each time `compute-gf-applier` is invoked, so the production of the specialized code is much slower than in the hand-coded case. We can solve this problem by re-staging the partial evaluation process itself--that is, analyzing `apply-gf` once to determine what optimizations will be possible when `(methods gf)` is available. Numerous implementations of the necessary analysis exist, and are called "Binding Time Analysis" in the literature [4]. Second, because existing partial evaluators are not well-integrated with compilers, we must compile the specialized code explicitly. With better integration similar to that of [1], this will not be a problem.

Using a partial evaluator as a staging tool provides two benefits: convenience, since the programmer doesn't have to manually transform the code, and correctness, because a correct partial evaluator will perform only semantics-preserving transformations. The combination of these benefits should allow for more staging in runtime meta-code implementations.

[1] Because their base language implementation lacked runtime compilation, the CLOS implementors were restricted to creating multiple lexical closures of the same code body. Thus, the specialized variants could differ only in the precomputed values in their environments, but were forced to share common code bodies. This meant that some operations, e.g. conditionals, could not be folded out. However, the extension to producing specialized code bodies is clear (indeed, it is presented as an exercise in [6]). Note, however, that regardless of whether closures or specialized code are generated, the code performing this generation is constructed manually.

[2] Specifically, we expect that the calls to `dolist` and `apply-method` on p. 127 of [6] would be inlined and optimized.

```
(define (foo x)
  (+ (car x) (cdr x)))

(define (bar a b
  (foo (cons a b))

(define (baz a b)
  (foo (cons a b))
```

Figure 2. Simplified example of the need for coordination at compile time. In the absence of specialization, all three functions must agree on the representation used by the operator `cons`. That is, `bar` and `baz` either must implement cons identically, or must provide tagging so that `foo` can perform dispatch at runtime. If partial evaluation is allowed to construct two variants of `foo`, the problem is solved.

3. Compile-time Meta-code

In the runtime case, we saw that partial evaluation could speed up programs by reducing the amount of meta-code executed per base-level instruction. In the case of compile-time meta-architectures (e.g., [8]), such reduction is unnecessary, since there *are* no meta-operations at runtime. However, partial evaluation can still be of use, by allowing the compile-time meta-code to make better implementation decisions.

One major difference between run-time and compile-time meta-architectures concerns coordination of decision making. In a runtime system, the meta-code is free to implement a particular base-level operation differently for each runtime context in which it is encountered. A compile-time system, however, must produce a single implementation that suffices for all anticipated runtime contexts in which the base-level operation being compiled will be invoked. For example, all of a function's call sites must agree on how the arguments to the function are passed, and on how they are represented. This need for a common representation is typically solved in one of two ways, both of which can hurt performance.

- Force the representations to be self-describing (in the sense of carrying tag bits or class membership information), so that the recipients of data will know how to perform operations on it. This results in extra dispatching operations when the program runs, hurting performance.
- Force all communicating parts of the implementation to agree on a single representation for the information being communicated. This can make it more difficult for the meta-program to choose locally-optimized implementations of base-level constructs (e.g., implementing `cons` differently in procedures `bar` and `baz` of Figure 2) if those implementations must also be manipulated by other parts of the program (e.g., `foo`). The meta-code would have to balance the benefits of local optimizations with the costs of coercing them to a common form.

Partial evaluation can help dramatically in both of the above situations by introducing polyvariance. Specializing a procedure into a different variant for each of its invocation contexts frees those contexts from having to agree on representations, eliminating the need for either runtime dispatch, sub-optimal common representations, or coercions. For example, in the code of Figure 2, if we create a separate version of `foo` for each of its call sites, then `bar` and `baz` would be free to implement `cons` pairs in different ways. In this case, we are partially evaluating the base code rather than the meta-code, and are speeding up the base code not by eliminating meta-operations, but by allowing the meta-code to make better implementation decisions.

Of course, nothing is keeping the meta-code from performing this analysis and introducing the polyvariance itself. What makes "black box" partial evaluation particularly attractive is that it lets the meta-code author (or user overriding a meta-code method in a particular location) spend far less effort worrying about certain kinds of coordination issues, such as those concerning implementation compatibility of procedure arguments. Of course, other coordination issues (such as the interactions described in Section 5 of [8]), must still be resolved explicitly by the meta-code.

4. Other considerations

We have been discussing potential performance improvements due to partial evaluation of both base- and meta-level programs, without treating the additional difficulties inherent in partially evaluating multi-level programs. Partial evaluators operate by abstract execution of the program under some domain of approximations to runtime values--this is possible only when the partial evaluator, in some sense, knows the meaning of the primitives it is simulating. In the runtime case, this is always possible---we can always simulate a base-level operation by simulating the meta-operations that implement it. However, as has been noted before [10] the need to operate at such a low level can make partial evaluation much more difficult. If, for example, the meta-code implements generic functions in several semantically equivalent ways (with only performance differences between the implementations), the partial evaluator is far more likely to do a good job if it had a notion of "generic function" and its properties, than if it had to analyze each `cons` and `setq` operation in each of the implementations.

The situation is even worse when we consider compile-time meta-architectures, since at the time the meta-code wishes to specialize a base-level procedure, it may not have yet chosen implementations for all of the operations in that procedure---in particular, it may be performing the specialization merely to see it creates the opportunity for a local optimization. In this case, the partial evaluator will be unable to simulate the base-level operations unless it is given some implementation-independent information about their semantics.

This suggests the use of a meta-protocol architecture for the partial evaluation process itself. That is, in addition to providing meta-code that determines how a base-level operation is to be implemented, we could also provide meta-code that determines how it will be partially evaluated. Were this to be done, we would have, in essence, a synergy in which the very method used to improve the performance of meta-level systems would be improved by the use of reflective concepts.

5. Related work

Introductions to partial evaluation, its capabilities and implementations, and further references, can be found in [4,10]. Also of interest are the techniques of customization [1] and procedure cloning [2], which can be viewed as partial evaluation under another name. In the reflection community, both [7] and [11] suggest partial evaluation as a performance enhancement technique. We believe that the a compiler-based implementation of ABCL/R [9] also uses (or will use) partial evaluation techniques. As we noted above, the CLOS MOP [6] gains much of its performance via a staging technique that amounts to a form of manual partial evaluation. The use of explicit meta-code is common in partial evaluation, though it is usually in a a fairly static, declarative form (e.g. annotations on the source program). However, recent work appears to be moving away from meta-annotations toward more monolithic, automated systems based on complex analyses [3,5]. Factoring out such analyses into the meta-level might both simplify them, and allow greater user intervention in PE.

6. Conclusion

Partial evaluation can be used to improve the performance of both run-time and compile-time meta-level architectures. In particular, it can help automate staging decisions in run-time systems, reducing the number of meta-operations performed per base operation. Improved locality due to the introduction of polyvariance can allow compile-time meta-code to choose more efficient runtime implementations while eliminating dispatch and coercion costs. Finally, meta-level architectures may themselves be used to improve the partial evaluation process.

These benefits will not appear immediately. In this author's view, the state of the art in partial evaluation is still rather weak with respect to imperative and concurrent constructs, which are of ten used in meta-code. Also, the meta-architecture community is still in a "defining" stage, in which the search for general principles and functionality appear to be more important issues than performance. It is the author's hope that the approaches suggested in this paper will become more realistic in the near future.

References

[1] Chambers, C. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Languages.* Ph.D. thesis, Stanford University, 1992. Published as technical report STAN-CS-92-1420.

[2] Cooper, K. D., Hall, M. W., and Kennedy, K. Procedure Cloning. In *IEEE International Conference on Computer Languages*, Oakland, CA, 1992.

[3] Holst, C. K. Finiteness Analysis. In *Functional Programming Languages and Computer Architecture* (1991), ACM, Springer-Verlag.

[4] Jones, N. D., Sestoft, P., and Sondergaard, H. *Partial Evaluation and Automatic Program Generation*, 1993.

[5] Katz, M. and Weise, D. Towards a new perspective on partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Directed Program Manipulation*, San Francisco, CA, 1991. Proceedings available as YALEU/DCS/RR-909.

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

[7] Kiczales, G. Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-Level Architectures* (1992), pp. 1-11.

[8] Lamping, J. Kiczales, G., Rodriguez Jr., L. H., and Ruf, E. An architecture for an open compiler. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-Level Architectures* (1992), pp. 95-106.

[9] Matsuoka, S., Watanabe, T., and Yonezawa, A. Hybrid group reflective architecture for object-oriented concurrent programming. In *European Conference on Object-Oriented Programming* (1991), pp. 231-250.

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