Multi-Stage Specialization with Relative Binding Times*

Mark Leone
Indiana University

Peter Lee
Carnegie Mellon University

Technical Report #497
Computer Science Department, Indiana University

November 1997

Abstract

Programming systems that generate code at run time offer unique opportunities for specialization. Dynamic specialization can exploit run-time values that are not available at compile time, yielding code that is superior to statically optimal code. Unfortunately, conventional formulations of binding-time analysis prove overly restrictive in such a setting. The values computed by specialized procedures are classified as dynamic, which prevents a useful form of *multi-stage specialization*. We propose a simple notion of *relative binding times* that allow multiple stages of specialization to be realized in a two-level lambda calculus.

1 Dynamic Specialization and Binding-Time Analysis

Most partial evaluators perform specialization at compile time. However, recent research has demonstrated that it is sometimes profitable to perform specialization at later stages, such as run time [8, 7, 2, 4, 6]. Some implementations of dynamic specialization rely on a binding-time analysis (BTA) to determine how to specialize a program when some of its variables take on fixed values. BTA requires an initial *binding-time division* that classifies program inputs as static or dynamic. From this initial division, the BTA translates a program into a *two-level language* in which every subexpression is annotated with its binding time.

Unfortunately, conventional BTA prevents dynamic specialization from exploiting many opportunities for optimization. This is most easily illustrated with an example:

*This research was partially supported by the Defense Advanced Research Projects Agency ITO under the title "The Fox Project: Advanced Language Technology for Systems Software," DARPA Order No. 8313, issued by ESC/ENS under Contract No. F19628-95-C-0050, and by the National Science Foundation under grant number CCR-9711269. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the the Defense Advanced Research Projects Agency, the National Science Foundation, or the U.S. Government.

Suppose a merchant wants to provide faster service to "big spenders" whose records indicate recent spending exceeding some threshold. Determining whether a customer is a big spender can be accomplished efficiently by first querying the customer database for a list of big spenders and then searching this list on every transaction:[1]

```
big-spender? threshold db key =
  let big-spenders = query `(> spent ,threshold) db
  in
      member big-spenders key
  end
```

This code presents two opportunities for performing specialization at run time: the `query` procedure can be specialized to the dynamically constructed search query, eliminating substantial interpretive overhead, and the `member` predicate can be specialized to the list of big spenders. (The efficacy of specializing a membership predicate at run time has been previously demonstrated [7].)

However, a conventional BTA does not allow both of these opportunities to be realized. Specializing the `query` procedure requires `threshold` to be static and `db` to be dynamic; this results in the classification of `big-spenders` as dynamic, which prevents `member` from being specialized.

We believe that a wide range of real-world programs exhibit similar opportunities for *multi-stage specialization*, where the "dynamic" results computed by dynamically specialized code can be employed in successive specializations. For example, common library routines like matrix multiplication benefit from run-time specialization [7], and it is easy to imagine that user code could benefit from being specialized to values computed by library code.

We propose a simple notion of *relative binding times* that allows multiple stages of specialization to be realized in a calculus with just two levels. Unlike the alternative approaches, in which an arbitrary number of "absolute" binding times indicate precisely the stage at which values become available (see Section 4), relative binding times assert only that certain values will become available before or after other values. As a demonstration of the approach, we have implemented a system called Fabius, which compiles a two-level calculus with relative binding times into highly efficient generating extensions that emit native code at run time.

2 A Calculus of Relative Binding Times

In this section we describe a two-level lambda calculus with relative binding times. Limited space prevents a complete formal description, but we shall endeavor to describe the essence of our approach. We shall abandon the use of "static" and "dynamic" as binding times. Instead, we use two *relative* binding times, "early" and "late."

[1] This code employs a back-quoted expression to construct a search query that compares the "`spent`" field of each record in the database with the given spending threshold.

As usual, binding times guide specialization. Early expressions evaluate fully, while late expressions evaluate to residual code:[2]

$$\frac{e_1 \hookrightarrow \lambda x\,.\,e \quad e_2 \hookrightarrow v \quad [v/x]e \hookrightarrow v'}{e_1 \mathbin{@} e_2 \hookrightarrow v'} \qquad \frac{e_1 \hookrightarrow e_1' \quad e_2 \hookrightarrow e_2'}{e_1 \mathbin{\underline{@}} e_2 \hookrightarrow e_1' \mathbin{@} e_2'} \qquad \underline{x} \hookrightarrow x$$

However, we adopt an unconventional evaluation strategy for λ-expressions. An "ordinary" λ-expression self-evaluates, with no reduction occurring in its body, but certain λ expressions, denoted $\widehat{\lambda} x\,.\,e$, are specialized to the values of their early free variables when evaluated:[3]

$$\lambda x\,.\,e \hookrightarrow \lambda x\,.\,e \qquad \frac{e \hookrightarrow e'}{\widehat{\lambda} x\,.\,e \hookrightarrow \lambda x\,.\,e'}$$

These evaluation rules are sensible only under certain conditions. For example, occurrences of x in the body of $\widehat{\lambda} x\,.\,e$ must be annotated as late. (Otherwise the evaluation of e would fail.) We therefore require that terms be *consistently annotated*, as defined by a judgment of the form $\Gamma \vdash e : bt$ where bt is a binding time and Γ is a context mapping variables to binding times. A conventional definition of consistency provides a good starting point:

$$\frac{\Gamma, [x \mapsto \text{early}] \vdash e : \text{early}}{\Gamma \vdash \lambda x\,.\,e : \text{early}} \qquad \frac{\Gamma, [x \mapsto \text{late}] \vdash e : \text{late}}{\Gamma \vdash \underline{\widehat{\lambda}} x\,.\,e : \text{late}} \qquad \Gamma \vdash x : \Gamma(x)$$

$$\frac{\Gamma \vdash e_1 : \text{early} \quad \Gamma \vdash e_2 : \text{early}}{\Gamma \vdash e_1 \mathbin{@} e_2 : \text{early}} \qquad \frac{\Gamma \vdash e_1 : \text{late} \quad \Gamma \vdash e_2 : \text{late}}{\Gamma \vdash e_1 \mathbin{\underline{@}} e_2 : \text{late}} \qquad \frac{\Gamma \vdash e : \text{early}}{\Gamma \vdash \underline{\text{lift}}\ e : \text{late}}$$

One special case merits attention, however: what is the binding time of an expression $\widehat{\lambda} x\,.\,e$ that contains no late free variables? The evaluation rule for $\widehat{\lambda}$ indicates that such a procedure will be specialized to the values of its early free variables, yielding a closed procedure. In a departure from conventional BTA, we claim that it is sensible for such an expression to be annotated as early:

$$\frac{\Gamma, [x \mapsto \text{late}] \vdash e : \text{late} \qquad y \in FV(\widehat{\lambda} x\,.\,e) \Rightarrow \Gamma(y) = \text{early}}{\Gamma \vdash \widehat{\lambda} x\,.\,e : \text{early}}$$

The rationale for this rule is simple: an expression should be annotated with an early binding time if its value depends solely upon early values. The above evaluation rules clearly indicate that the value of a lambda expression does not depend on the value of its argument, so it is sensible for the binding times of procedures and arguments to be unrelated.

This rule leads to some examples that may be counterintuitive until they are examined with an eye towards performing specialization at run time. For example, a $\widehat{\lambda}$ expression with no late free variables can be applied to an early value, yielding an early result:

$$\lambda m\,.\,(\widehat{\lambda} n\,.\,\underline{\text{lift}}\ m \mathbin{\underline{+}} \underline{n}) \mathbin{@} (m{+}1) : \text{early} \tag{1}$$

[2]Late expressions are annotated with underscores. Applications are explicitly indicated with "@" symbols to simplify underscoring.

[3]The attentive reader may notice that this rule precludes unfolding because it does no alpha conversion. We have done so deliberately only to simplify the presentation.

When this expression is applied to a constant c, the $\widehat{\lambda}$ expression will be specialized on this value. As the evaluation rules above indicate, however, evaluation will continue by applying the specialized code:

$$\begin{aligned}(\lambda m\,.\,(\widehat{\lambda} n\,.\,\underline{\text{lift}}\ m \mathbin{\underline{+}} \underline{n})\ @\ (m{+}1))7 &\rightarrow (\widehat{\lambda} n\,.\,\underline{\text{lift}}\ 7 \mathbin{\underline{+}} \underline{n})\ @\ (7{+}1)\\ &\rightarrow (\lambda n\,.\,7{+}n)\ @\ 8\\ &\rightarrow 7{+}8\\ &\rightarrow 15\end{aligned}$$

In this example, it is not necessary to build a residual application after specializing the $\widehat{\lambda}$ expression because it contains no late free variables. Instead, the specialized code can be immediately applied to an early argument, yielding an early result.

The above example leads directly to an example that exhibits multi-stage specialization (let E be equation 1):

$$\begin{aligned}(\lambda a\,.\,\widehat{\lambda} b\,.\,\underline{\text{lift}}\ a \mathbin{\underline{*}} \underline{b})\ @\ E &\rightarrow^{+} (\lambda a\,.\,\widehat{\lambda} b\,.\,\underline{\text{lift}}\ a \mathbin{\underline{*}} \underline{b})\ @\ 15\\ &\rightarrow \widehat{\lambda} b\,.\,\underline{\text{lift}}\ 15 \mathbin{\underline{*}} \underline{b}\\ &\rightarrow \lambda b\,.\,15 * b\end{aligned}$$

Similarly, we can solve the example problem posed in Section 1 by defining both `query` and `member` to be lambda expressions of form $(\lambda x\,.\,\widehat{\lambda} y\,.\,e)$ and annotating the remaining code as follows:

```
big-spender? = λ threshold . λ db . λ̂ key .
  let big-spenders = query @ `(> spent ,threshold) @ db
  in
      member @ big-spenders @ key
  end
```

The list of `big-spenders` is early, even though it is computed by a dynamically specialized procedure, because `query` is closed and both `threshold` and `db` are early. It can therefore be used to specialize the `member` procedure.

The fact that the binding times of formal and actual parameters are unrelated yields a novel form of polyvariance that allows many more opportunities for dynamic specialization. It is even possible to pass late values as early arguments. Consider for example, these functions f and g:

$$\begin{aligned}f &= \lambda m\,.\,(\widehat{\lambda} n\,.\,\underline{\text{lift}}\ m \mathbin{\underline{+}} \underline{n})\\ g &= \lambda a\,.\,(\widehat{\lambda} b\,.\,\underline{\text{lift}}\ a \mathbin{\underline{+}} (\underline{\text{lift}}\ f\ \underline{@}\ \underline{b}))\end{aligned}$$

When g is initially invoked with a value for a, a specialized function g_a is generated. When g_a invoked with a value for b, a second stage of specialization occurs, yielding f_b. In practice, the dynamically generated code g_a contains a call to the statically derived generating extension for f. This is sensible in our framework because the only requirement is that the early and late arguments are *locally consistent*. In essence, we have discarded the global consistency requirement of conventional BTA in favor of a much simpler, and also more flexible, local annotation scheme.

3 The Fabius System

We have implemented our approach in a prototype compiler called FABIUS[4] [8, 9, 7]. It compiles a pure, first-order subset of ML that includes reals, vectors, and user-defined datatypes into native MIPS code. A novel formulation of *cogen* is employed to compile certain functions (corresponding to the $\widehat{\lambda}$ expressions used in this paper) into generating extensions that produce native residual code with extremely low overhead. By avoiding the construction of intermediate syntax trees, the cost of specialization and code generation is reduced to approximately 6 to 10 cycles per generated instruction. Besides the usual transformations performed by partial evaluators such as Similix [1], optimizations such as instruction selection, simple reduction of strength, and some register optimizations are also performed by the generating extensions. Our benchmarks [9, 7] demonstrate substantial speedups in application domains ranging from symbolic programs (such as regular-expression matching, cellular automata, etc.) to numerical applications (matrix multiply, conjugate gradient) to operating systems (network packet filtering).

4 Related Work

Multi-level languages [5, 3, 11, 10, 12] allow multi-stage specialization by permitting an arbitrary number of binding times. In one such framework [5], expressions are annotated with numbered binding times. In our example, `threshold` has a binding time of "0", `db` has a binding time of "1", and `key` has a binding time of "2", yielding the following multi-level program:

```
big-spender? threshold0 db1 key2 =
  let1 big-spenders1 = query @0 `(> spent ,threshold)0 @1 db1
  in
      member @1 big-spenders1 @2 key2
  end
```

Multi-level languages lead naturally to *multi-level generating extensions* [5]. For example, the generating extension for `big-spender?` specializes the `query` procedure and then creates a *specialized* generating extension parameterized by `db`.

However, multi-level generating extensions are impractical when specialization is performed at run time, because the cost of creating specialized generating extensions is unlikely to be repaid. As we have demonstrated, most of the benefit of multi-stage specialization can be achieved cheaply and easily with two-level generating extensions.

5 Conclusions

We have developed a simple and effective approach to multi-stage specialization based on the notion of "relative binding times". We have implemented a prototype system called FABIUS

[4]The compiler is named after Quintus Fabius Maximus, who was a Roman general best known for his defeat of Hannibal in the Second Punic War. His primary strategy was to delay confrontation; repeated small attacks eventually led to victory without a single decisive conflict.

and shown experimentally that this approach is effective [7]. We are currently implementing a successor to FABIUS that compiles higher-order procedures into abstract machine code for a novel architecture called the Two-Level Abstract Machine, which simplifies experimentation while realistically modeling relevant characteristics of real architectures.

In future work we plan to consider run-time issues such as garbage collection of dynamically generated code. Finally, we plan to investigate the use of modal and temporal type systems to provide a systematic annotation scheme for programmer guidance of dynamic specialization.

References

[1] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(2):151–195, September 1991.

[2] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, January 1996.

[3] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 258–270, January 1996.

[4] Scott Draves. Compiler generation for interactive graphics using intermediate code. In *Dagstuhl Workshop on Partial Evaluation (LNCS1110)*, 1996.

[5] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions. In *Programming Languages, Implementations, Logics and Programs (PLILP'95),* volume 1181 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[6] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. Annotation-directed run-time specialization in C. In *PEPM 97 Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, June 1997.

[7] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 137–148, Philadelphia, Pennsylvania, May 1996.

[8] Mark Leone and Peter Lee. Lightweight run-time code generation. In *PEPM 94 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106. Technical Report 94/9, Department of Computer Science, University of Melbourne, June 1994.

[9] Mark Leone and Peter Lee. A declarative approach to run-time code generation. In *WCSSS'96 Workshop on Compiler Support for System Software*, pages 8–17, February 1996.

[10] Flemming Nielson and Hanne Riis Nielson. Prescriptive frameworks for multi-level lambda-calculi. In *PEPM 97 Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, June 1997.

[11] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *PEPM 97 Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, June 1997.

[12] Philip Wickline, Peter Lee, Frank Pfenning, and Rowan Davies. Modal types as staging specifications for run-time code generation. *Submitted to Computing Surveys Symposium on Partial Evaluation*, 1997.