Dynamic Specialization in the Fabius System

Mark Leone
Computer Science Department, Indiana University
and
Peter Lee
School of Computer Science, Carnegie Mellon University

Opportunities for specialization abound during program execution. *Dynamic specialization* exploits run-time values that are not available at compile time, yielding code that is often superior to statically optimal code. This paper surveys the design of the FABIUS compiler, which achieves dynamic specialization with low overhead through the use of generating extensions that create native residual code. One novel aspect of FABIUS is its ability to achieve multiple stages of dynamic specialization: the execution of dynamically specialized code yields values that can be used in successive rounds of specialization. We demonstrate that *decoupling* binding times at procedure call boundaries allows multi-stage specialization to be expressed in a two-level language.

Categories and Subject Descriptors: D.3.4 [**Software**]: Processors—*Compilers*

General Terms: Languages, Design

Additional Key Words and Phrases: Run-time code generation, Multi-stage specialization, Binding-time analysis, Two-level languages, Partial evaluation

1. DYNAMIC SPECIALIZATION

Most partial evaluators perform specialization at compile time. However, recent research has demonstrated that it is sometimes profitable to specialize code at run time [Lee and Leone 1996; Consel and Noël 1996; Draves 1996; Grant et al. 1997]. For example, many Unix kernels contain a simple interpreter that securely executes user-level packet-filtering code; this interpreter can be specialized each time a new packet filter is installed [Engler et al. 1995; Leone and Lee 1996].

To appear in *Computing Surveys Symposium on Partial Evaluation*, 1998.
Address: Indiana University, Bloomington, IN 47405, and Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213-3891
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.

Of course, the overhead of invoking a full-fledged partial evaluator at run time can be substantial. The use of *generating extensions* shifts much of this overhead to compile time, but the cost of run-time code generation must also be addressed. Most partial evaluators perform source-to-source program transformations and yield generating extensions that produce residual *source* code. Such generating extensions are not suitable for dynamic specialization because the overhead of dynamically compiling or interpreting residual source code often exceeds the benefits of performing specialization at run time. However, as many researchers have observed, residual code need not be expressed in the same language as the subject program.

2. THE FABIUS COMPILER

Fabius was the first system to demonstrate the feasibility of deriving generating extensions that generate *native* residual code [Leone and Lee 1994]. Fabius compiles a subset of ML into code that performs dynamic specialization with extremely low overhead. A programmer can use a curried definition to introduce a binding-time division that is *local* to a single procedure. For example, consider the following ML procedure, which computes the inner product of two lists of numbers:

```
fun dotprod [] [] = 0
  | dotprod (x::xs) (y::ys) = (x * y) + (dotprod xs ys)
```

Fabius compiles this into a generating extension, parameterized by the first list, that generates native code for a procedure parameterized by the second list. Such specialized code generators are highly efficient, requiring an average of only six cycles per generated instruction [Lee and Leone 1996].

3. LOCAL BINDING-TIME DIVISIONS

The concept of a local binding-time division merits further discussion. Most offline specializers are parameterized by an initial division that classifies program inputs as either *static* or *dynamic*. Binding-time analysis (BTA) extends this initial division to a *global* division of all computed values. Dynamic specialization, on the other hand, is not driven by a division of program inputs; instead, one can view binding times as an indication of the order in which values are computed. For example, the first argument of the `dotprod` procedure might might be considered static because it is computed prior to the execution of a loop in which the second argument varies. To avoid confusion, we shall use the terms *early* and *late* instead of *static* and *dynamic*, since all the values under consideration are dynamic.

In some situations it is useful to introduce a local binding-time division where none would otherwise exist. For example, consider a matrix multiplication procedure parameterized by two matrices (`m1` and `m2`) that are both early. The multiplication might begin by calling `dotprod` with a row from `m1` and a column from `m2`. As we have previously demonstrated [Leone and Lee 1994], it is often profitable to dynamically specialize the `dotprod` procedure, since a single row from `m1` is used to compute multiple dot products. In order to specialize `dotprod` we must regard its first argument as early and its second argument as late, which leads to the following binding-time annotations:

$$\begin{array}{l}\texttt{fun dotprod [] [] = }\underline{\mathrm{lift}}\,0\\ \quad\texttt{| dotprod (x::xs) (}\underline{\texttt{y::ys}}\texttt{) = (}\underline{\mathrm{lift}}\,x\ \underline{*}\ \underline{\texttt{y}}\texttt{)}\ \underline{+}\ \texttt{(dotprod xs }\underline{\texttt{ys}}\texttt{)}\end{array}$$

Now consider the binding time of the result returned by `dotprod`: the result is either lift 0 or the result of a late addition, so it will be a late value. However, `dotprod` is called with vectors selected from two early matrices, so the result *should* be early. To resolve this dilemma, FABIUS *decouples* binding times at procedure call boundaries:

—The binding times of actual arguments do not constrain the binding times of formal parameters, and vice versa. The binding times of formal parameters are determined solely by the syntactic form of the procedure definition: the formals are all early unless the definition is curried.

—The binding time of a call to a closed procedure is the least upper bound of the binding times of the actual arguments.

This decoupling allows local binding-time divisions to be introduced where none would otherwise exist, and it allows values to assume completely different binding times in different contexts.

The key to a sound interpretation of decoupled binding times is the fact that specialization and execution of residual code may be interleaved. Recall that FABIUS compiles the `dotprod` procedure into a generating extension that takes the early argument (`xs`) and generates code parameterized by the late argument (`ys`). Suppose `v1` is an early value and `v2` is a late value:

—`dotprod v1 v2` is compiled into an early call to the generating extension, followed by code that constructs a residual call to the generated code.[1] The overall result is late because it depends on `v2`.

—`dotprod v1 v1` is compiled into an early call to the generating extension, followed by an early call to the generated code (i.e. the residual code is immediately executed). The overall result is early because it depends solely on early values.

—`dotprod v2 v2` is compiled into code that constructs a residual call to the generating extension and a residual call to the generated code. The result is late.

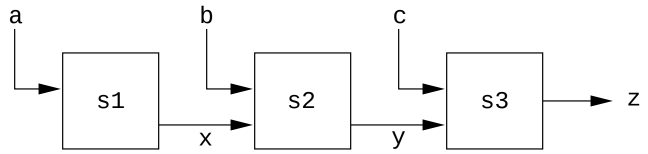
—`dotprod v2 v1` is compiled into code that constructs a residual call to the generating extension and a residual call (in which `v1` is lifted) to the to the generated code. The result is late.

Despite the fact that the binding times of the actual arguments do not match the binding times of the formal parameters, all of these applications are sensible.

4. MULTI-STAGE SPECIALIZATION

Large software systems often exhibit many stages of computation, and the values computed in early stages are often fixed for the duration of later stages. FABIUS is capable of exploiting such staging through *multi-stage specialization*, which uses the values computed by dynamically specialized code in successive rounds of specialization. For example, consider a program `P` consisting of three stages `s1`, `s2`, and `s3` that are parameterized by inputs `a`, `b`, and `c`:

[1]In some cases such procedure calls are unfolded, eliminating the residual call to the generated code.



Suppose we wish to specialize each later stage to the result of the preceding stage before processing the remaining input. This can be achieved as follows:

```
fun P(a,b,c) = let x = s1 a
                   y = s2 x b
               in s3 y c
```

When compiled by FABIUS, this code behaves as follows:

—`s1` is called with `a`, yielding `x`.

—`s2` is specialized to `x`, yielding `s2′`. `s2′` is then called with `b`, yielding `y`.

—`s3` is specialized to `y`, yielding `s3′`. `s3′` is then called with `c`, yielding `z`.

In this example, all of the variables in `P` have identical binding times.[2] As discussed in the previous section, the currying of stage `s2` introduces a local binding-time division in which `b` is treated as a late value, but a decoupling of binding times allows `y` to have an early binding time.

Despite employing only two binding times, FABIUS can achieve an arbitrary number of stages of specialization. Its binding times are neither global nor absolute; instead they are local and relative.

5. RELATED WORK

Tempo is a partial evaluator for C that can be used for both static and dynamic specialization [Consel and Noël 1996]. It employs a sophisticated BTA that addresses a wide range of language features (including side effects, aliasing, etc.), but to the best of our knowledge the BTA does not permit multi-stage specialization.

Multi-level languages [Glück and Jørgensen 1995; Davies and Pfenning 1996; Taha and Sheard 1997; Nielson and Nielson 1997; Wickline et al. 1997] provide an alternative to our approach of expressing multi-stage specialization in a two-level calculus. Multi-level languages permit the derivation of *multi-level generating extensions* [Glück and Jørgensen 1995], which create specialized generating extensions. This might be useful for applications in which dynamic specialization incurs a substantial amount of interpretive overhead. The overhead of dynamically generating a specialized generating extension is substantial, however, and we believe that most of the benefit of multi-stage specialization can be realized more easily and efficiently with two-level generating extensions.

[2]This example does not illustrate another useful form of multi-stage specialization that is enabled by the fact that procedures can be specialized to late values.

6. CONCLUSIONS AND FUTURE WORK

We have implemented a compiler called FABIUS that compiles a subset of ML into code that generates specialized code at run time. The overhead of dynamic specialization is reduced through the use of generating extensions that emit native code. FABIUS employs a binding-time analysis that permits multi-stage specialization to be realized within a two-level language. A novel form of polyvariance is achieved by decoupling the binding times of formal parameters from those of actual arguments, which allows a value to assume different binding times in different contexts.

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 investigate a wider range of dynamic optimizations on both high- and low-level intermediate representations [Leone and Dybvig 1997]. Finally, we plan to investigate the use of modal and temporal type systems [Davies and Pfenning 1996; Wickline et al. 1997] to provide a systematic annotation scheme for programmer guidance of dynamic specialization.

REFERENCES

CONSEL, C. AND NOËL, F. 1996. A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (January 1996), pp. 145–156.

DAVIES, R. AND PFENNING, F. 1996. A modal analysis of staged computation. In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (January 1996), pp. 258–270.

DRAVES, S. 1996. Compiler generation for interactive graphics using intermediate code. In *Dagstuhl Workshop on Partial Evaluation (LNCS1110)* (1996).

ENGLER, D. R., WALLACH, D., AND KAASHOEK, M. F. 1995. Efficient, safe, application-specific message processing. Technical Memorandum MIT/LCS/TM533 (March), MIT Laboratory for Computer Science.

GLÜCK, R. AND JØRGENSEN, J. 1995. Efficient multi-level generating extensions. In *Programming Languages, Implementations, Logics and Programs (PLILP'95),* volume 1181 of *Lecture Notes in Computer Science* (1995). Springer-Verlag.

GRANT, B., MOCK, M., PHILIPOSE, M., CHAMBERS, C., AND EGGERS, S. J. 1997. Annotation-directed run-time specialization in C. In *PEPM 97 Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (June 1997).

LEE, P. AND LEONE, M. 1996. Optimizing ML with run-time code generation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, May 1996), pp. 137–148.

LEONE, M. AND DYBVIG, R. K. 1997. Dynamo: A staged compiler architecture for dynamic program optimization. Technical Report 490 (September), Computer Science Department, Indiana University.

LEONE, M. AND LEE, P. 1994. Lightweight run-time code generation. In *PEPM 94 Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (June 1994), pp. 97–106. Technical Report 94/9, Department of Computer Science, University of Melbourne.

LEONE, M. AND LEE, P. 1996. A declarative approach to run-time code generation. In *WCSSS'96 Workshop on Compiler Support for System Software* (February 1996), pp. 8–17.

NIELSON, F. AND NIELSON, H. R. 1997. Prescriptive frameworks for multi-level lambda-calculi. In *PEPM 97 Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (June 1997).

Taha, W. and Sheard, T. 1997. Multi-stage programming with explicit annotations. In *PEPM 97 Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (June 1997).

Wickline, P., Lee, P., Pfenning, F., and Davies, R. 1997. Modal types as staging specifications for run-time code generation. *Submitted to Computing Surveys Symposium on Partial Evaluation*.