Modal Types as Staging Specifications for Run-Time Code Generation

Philip Wickline
Peter Lee
Frank Pfenning
Rowan Davies
School of Computer Science, Carnegie Mellon University

Run-time code generation has been shown to be a powerful technique for exploiting specialization opportunities in many programs. We propose the use of source-level annotations from a modal λ-calculus to guide run-time code generation. This is in contrast to previous schemes which rely on *ad hoc* programmer annotations or binding-time analysis. Early experience with the approach is gained through a prototype system that compiles Mini-ML$^{\Box}$, a dialect of ML with modal types, into code for the CCAM, a Categorical Abstract Machine extended with support for run-time code generation.

Categories and Subject Descriptors: D.3.3 [**Software**]: Programming Languages—*Language Constructs and Features*

General Terms: Languages, Design

Additional Key Words and Phrases: Run-time code generation, Partial evaluation, Modal logic

---

1. RUN-TIME CODE GENERATION

A well-known technique for improving the performance of computer programs is to separate the computations into distinct stages so that the results of early computations can be profitably exploited by later computations. A particular realization of this technique is to arrange for compiled code to generate specialized code at run time. This approach, which is referred to as run-time code generation (RTCG),

---

Address: 5000 Forbes Avenue, Pittsburgh, PA 15213-3891

This research was supported in part by the National Science Foundation under grant #CCR-9619832, and by 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. 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 National Science Foundation, the Defense Advanced Research Projects Agency or the U.S. Government.

allows even low-level code optimizations to take advantage of values that are not known until run time. Previous work shows the efficacy of this approach in many situations [Pu et al. 1995; Engler and Proebsting 1994; Keppel et al. 1993].

To take a simple example, consider the following function (in ML-style syntax) which evaluates a given polynomial for a given base. For this function, the polynomial $a_0 + a_1x + a_2x^2 + \cdots a_nx^n$ is represented as the list $[a_0, a_1, a_2, \ldots, a_n]$.

```
type poly = int list;

val poly1 = [2, 3, 0, 2333];

(* val evalPoly : int * poly -> int *)
fun evalPoly (x, nil) = 0
  | evalPoly (x, a::p) =
    a + (x * evalPoly (x, p));
```

If this function were called many times with the same polynomial but different bases, it might be profitable to specialize it to the particular polynomial, in effect synthesizing an ML function that directly computes the polynomial rather than interpreting its list representation. One way that we can accomplish this is by transforming the code as follows.

```
fun specPoly (nil) =
    (fn x => 0)
  | specPoly (a::p) =
    let
        val polyp = specPoly p
    in
        fn x => a + (x * polyp x)
    end

val poly1target = specPoly poly1;
```

While `poly1target` is an improvement over the more general `evalPoly`, it is far from the fully specialized result we would like. Without support from the compiler, common source-level optimizations are not performed, such as unfolding of applications (to avoid the creation of closures). Furthermore, code-level optimizations cannot take advantage of the staging, for example in instruction selection (to avoid memory loads for early-stage values), register allocation (to take advantage of the fact that `specPoly` and `poly1target` can use the same registers), and elimination of early bounds checks (in the case of array-processing programs). Our goal is to exploit staged computation by efficiently performing such optimizations at run time.

2. DETERMINING PROGRAM STAGING

In order to perform RTCG, a compiler must understand how the program's computations are staged. Determining this staging information is no simple task. While automatic binding-time analysis has been used by some compilers (notably the

Tempo system [Consel and Noël 1996]), we are interested in developing a systematic method for programmer-supplied annotations to describe staging.

2.1 *ad hoc* Annotations

Annotation schemes used by working systems include that of ‘C [Engler et al. 1996], Fabius [Lee and Leone 1996], and the Lisp macros. These languages allow the programmer to communicate his intentions to the compiler in a relatively straightforward manner. Unfortunately, the Fabius annotation scheme is extremely simple, limiting the ability of the programmer to express staging decisions. The difficulty in ‘C and Lisp, on the other hand, is that they provide no direct way to ensure that the specified staging behavior is correct: programs can be written that will result in run-time errors. Possible errors include referencing a variable that is not yet available, and referencing variables which are no longer available.

2.2 Typed Annotations

In Davies and Pfenning [1996], it is proposed that the language Mini-ML$^\Box$, based on modal S4, be used to express the staging properties of programs. They show that binding-time correctness is equivalent to modal correctness by exhibiting a conservative embedding of Nielson and Nielson’s two-level functional language [Nielson and Nielson 1992] in Mini-ML$^\Box$.

In the next section we propose an interpretation of the modal “box” constructs as the locations of generating extensions for RTCG. We believe that using the modal source language has the following advantages:

—The programmer can express the desired staging directly to the compiler.

—This framework allows the programmer to verify the correctness of his staging intentions. Furthermore, this framework is useful for conceptualizing and discussing staging with other programmers.

—It complements the use of automatic staging through binding-time analysis. A compiler may well augment the staging requirement from a hand-staged program using some other means.

—It is appropriate for situations in which more than two stages are desired, such as Fabius-style multi-level specialization [Lee and Leone 1998].

As an example of an ML$^\Box$ program, we rewrite the `specPoly` function as follows:

```
type poly = (int $) list;

(* val compPoly : poly -> (int -> int) $ *)
fun compPoly (nil) =
    box (fn x => 0)
  | compPoly (a::p) =
    let
        boxed f = compPoly p
        boxed a' = a
    in
        box (fn x => a' + (x * f x))
    end
```

Fig. 1. $\lambda^{\Box}$ Syntax

Types	A, B	$::=$	$A \rightarrow B \mid \Box A \mid t$
Terms	M, N	$::=$	$x \mid \lambda x : A.M \mid MN$
			$\mid u \mid \text{box M}$
			$\mid \text{let boxed u = M in N}$
Contexts	Γ, Δ	$::=$	$\cdot \mid \Gamma, a : A$

Fig. 2. Typing rules for $\lambda^{\Box}$

$$\frac{x : A \text{ in } \Gamma}{\Delta; \Gamma \vdash x : A} \qquad \frac{\Delta; \Gamma \vdash M : A \rightarrow B \quad \Delta; \Gamma \vdash N : A}{\Delta; \Gamma \vdash MN : B}$$

$$\frac{\Delta; \cdot \vdash M : A}{\Delta; \Gamma \vdash \text{box M} : \Box\text{A}} \qquad \frac{\Delta; (\Gamma, x : A) \vdash M : B}{\Delta; \Gamma \vdash \lambda x.M : A \rightarrow B}$$

$$\frac{u : A \text{ in } \Delta}{\Delta; \Gamma \vdash u : A} \qquad \frac{\Delta; \Gamma \vdash M : \Box A \quad (\Delta, u : A); \Gamma \vdash N : B}{\Delta; \Gamma \vdash \text{let boxed u = M in N : B}}$$

```
val poly1' = [box 2, box 3, box 0, box 2333];
val codeGenerator = compPoly poly1';
val mlPolyFun = eval codeGenerator;
```

Here the box operator marks the introduction of a code generator, and the postfix type constructor \$ is the type of code generators. Thus the `compPoly` function takes a list of code generators for integers and transforms it into a code generator for a function that computes the value of the polynomial for a particular base.

3. $\lambda^{\Box}$

We briefly introduce the language $\lambda^{\Box}$ which is a simplification of Mini-ML$^{\Box}$ described in Davies and Pfenning [1996]. Although we present only $\lambda^{\Box}$ here because of space considerations, the compilation technique described in the next section extends easily to all SML constructs. Indeed, we have implemented a prototype compiler for large subset of core ML extended with the modal constructs.

3.1 Syntax

We use t, u, v to range over modally bound variables (i.e. those bound by let boxed constructions), x, y, z for non-modal variables, and a, b, c to range over either. Figure 1 presents the syntax of $\lambda^{\Box}$.

3.2 Typing Rules

The typing rules use two contexts: a modal context Δ and a non-modal context Γ. The typing rules are essentially the familiar ones for the λ-calculus plus the rules for "let boxed" and "box". The box rule ensures that there are no free non-modal variables in an expression of box type, which indicates that it is safe to generate code for these expressions. The let boxed rule simply augments the modal context.

Table 1. Reduction steps on the CCAM for various functions in the text

Computation	Reductions
`evalPoly` (47, `poly1`)	807
`specPoly poly1`	443
`poly1Target` 47	175
`compPoly poly1'`	553
`eval codeGenerator`	200
`mlPolyFun` 47	74

4. COMPILATION

We have developed an extension of the Categorical Abstract Machine (CAM), called the CCAM (Code CAM), which we use as a target for our $\mathrm{ML}^{\Box}$ compiler. The primary addition to the CCAM is a single parameterized instruction, emit, which "generates" its instruction parameter to a code context which the machine can later execute.

The compilation strategy interprets box expressions as code generators parameterized by their free modal variables (recall that boxed code has no free non-modal variables), which will be bound to other code generators. These code generators are then activated when a modal variable is referenced outside of a modal context. The resulting value can then be used as a normal program object.

Although CCAM code is rather abstract in comparison to native machine code, we can still observe performance improvements by counting the number of reductions performed by our CCAM simulator. Table 1 summerizes the observed reduction counts for the different versions of the polynomial evaluator, measured using our $\mathrm{ML}^{\Box}$ compiler and CCAM simulator.

Using our $\mathrm{ML}^{\Box}$ compiler and CCAM simulator, we observe the reduction counts in Table 1 for the different versions of the polynomial evaluator.

5. CONCLUSION

We have designed and implemented a compiler for the language $\mathrm{ML}^{\Box}$ which compiles box expressions into code generators. We believe $\mathrm{ML}^{\Box}$ offers safety and programming ease superior to current systems which use annotations. While our initial experiments are encouraging, further research is necessary to determine if the language is powerful enough to conveniently express a wide range of useful program stagings, and whether the low-level optimizations used by 'C and Fabius can be successfully reproduced by our system.

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* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (21–24 Jan. 1996), pp. 145–156.

Davies, R. and Pfenning, F. 1996. A modal analysis of staged computation. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (21–24 Jan. 1996), pp. 258–270.

Engler, D. R., Hsieh, W. C., and Kaashoek, M. F. 1996. 'C: A language for high-level, efficient, and machine-independent dynaic code generation. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (21–24 Jan. 1996), pp. 131–144.

Engler, D. R. and Proebsting, T. A. 1994. DCG: An efficient, retargetable dynamic code generation system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)* (October 1994), pp. 263–272. ACM Press.

Keppel, D., Eggers, S. J., and Henry, R. R. 1993. Evaluating runtime-compiled value-specific optimizations. Technical Report 93-11-02 (November), Department of Computer Science and Engineering, University of Washington.

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, 21–24May 1996), pp. 137–148.

Lee, P. and Leone, M. 1998. Multilevel run-time specialization in the Fabius system. *Submitted to ACM Computing Surveys 1998 Symposium on Partial Evaluation*.

Nielson, F. and Nielson, H. R. 1992. *Two-Level Functional Languages*. Cambridge University Press.

Pu, C., Autrey, T., Black, A., Consel, C., Cowan, C., Inouye, J., Kethana, L., Walpole, J., and Zhang, K. 1995. Optimistic incremental specialization: streamlining a commercial operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles* (3–6Dec. 1995), pp. 314–324. ACM.