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

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

General Terms: Languages, Design

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

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, 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 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 $[a_0, a_1, a_2, \ldots, a_n]$.

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. C533, 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.

```
type poly = int list;

(* 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 a 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.

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

val poly1 = [2, 2333];
val poly1target : int -> int = 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, optimizations at the level of machine code cannot easily take advantage of the staging, for example in instruction selection, register allocation, and elimination of bounds checks. Our goal is to exploit staged computation by efficiently performing such optimizations through run-time code generation.

An ordinary program may admit many different ways of staging its computation. Therefore, a successful run-time code generation system rests on two primary components: (1) a means to express the desired staging, and (2) a compiler and run-time system to take advantage of this information. In this paper we focus on the former, describing a simple, clean, and logically motivated language with internal means of expressing computation stages particularly well-suited for run-time code generation.

2. A MODAL LAMBDA-CALCULUS

We briefly introduce the language $\lambda^{\Box}$ which is a simplification of the explicit version of ML$^{\Box}$ described in Davies and Pfenning [1996]. Although we present only $\lambda^{\Box}$ because of space considerations, the interpretation of `code` constructors as generating extensions described here has been extended in our prototype implementation to most of Standard ML (without modules).

$\lambda^{\Box}$ arises from the simply-typed λ-calculus by adding a new type constructor "$\Box$". It is related to the modal logic S_4 by an extension of the Curry-Howard isomorphism, where $\Box A$ means "*A is necessarily true*". The syntax of $\lambda^{\Box}$ is described in figure 1.

$$
\begin{array}{llll}
\text{Types} & A, B & ::= & A \to B \mid \Box A \\
\text{Terms} & M, N & ::= & x \mid \lambda x.M \mid MN \\
 & & & \mid u \mid \texttt{code}\ M \\
 & & & \mid \texttt{let cogen}\ u = M\ \texttt{in}\ N \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A \mid \Gamma, u : A
\end{array}
$$

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

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

$$
\frac{\Delta; \cdot \vdash M : A}{\Delta; \Gamma \vdash \texttt{code}\ M : \Box A} \qquad \frac{\Delta; (\Gamma, x : A) \vdash M : B}{\Delta; \Gamma \vdash \lambda x.M : A \to B}
$$

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

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

2.1 `code` Expressions as Generating Extensions

For staging run-time code generation, we think of $\Box A$ as the type of generating extensions for code of type A. Generating extensions, or code generators, are created with the `code` construct. For example, $\vdash$ `code` $(\lambda x.x) : \Box(A \to A)$ is a generator which, when invoked, generates code for the identity function.

The elimination form for $\Box$, `let cogen` $u = M$ `in` N, binds a new *code variable*, u, to the generating extension for code M' returned by the evaluation of M. When a code variable u is found during code generation in a generating extension (i.e. a `code` expression) for N', the generating extension for M' to which u is bound emits M' into N', effectively performing the substitution $[N'/u]M'$.

On the other hand, if u is encountered during normal evaluation, the code generator to which it is bound will be invoked and the resulting code executed. For example, the following function `eval` of type $\Box A \to A$, activates its code generator argument, and then immediately executes the generated code, returning its result.

```
fun eval x = let cogen u = x in u end
```

2.2 Type Discipline

The typing rules use two contexts: a modal context Δ in which code variables are declared, and an ordinary context Γ declaring value variables. The typing rules are the familiar ones for the λ-calculus plus the rules for "`let cogen`" and "`code`".

The rule for `code` ensures that no value variable occurs free in a generating extension. The importance of this to run-time code generation is explained below. The `let cogen` rule simply places a code generator into the modal context.

2.3 Programming with ML$^{\Box}$

As an example of an ML$^{\Box}$ program, we rewrite the `specPoly` function as follows (in ML$^{\Box}$, we use the postfix type constructor `$` to represent the $\Box$ type):

```
(* val compPoly : poly -> (int -> int) $ *)
```

```
fun compPoly (nil) = code (fn x => 0)
  | compPoly (a::p) =
    let cogen f = compPoly p
        cogen a' = lift a
    in code (fn x => a' + (x * f x)) end

val mlPolyFun : int -> int = eval (compPoly poly1);
```

The `compPoly` function takes a representation of a polynomial and returns a code generator for a function which computes that polynomial. It uses the built-in operator `lift`, which can be viewed as a function of type $A \to \Box A$. `lift` returns a code generator which directly reproduces either its argument or a pointer to it, depending on its type. While `lift` is definable for observable types in the pure ML$^{\Box}$-calculus, `lift` at function types has no direct logical meaning. It has nonetheless a sensible operational interpretation, and is probably required for any realistic implementation.

2.4 $\lambda^{\Box}$ and Run-time Code Generation

$\lambda^{\Box}$ exhibits a number of features which we believe are particularly useful for staging programs for run-time code generation:

(1) $\lambda^{\Box}$ allows the programmer to specify when and where they want code generation to occur. This ability makes staging a local property of code, meaning functions can be written without knowing the context in which they will be called. In contrast, traditional partial evaluation usually views staging as a global property of programs, assuming a fixed number of stages (most often two) imposed upon the program from the outside.

(2) $\lambda^{\Box}$ ensures that programs will not "go wrong" at run time, i.e., all binding-time errors will manifest themselves as static type errors.

(3) $\lambda^{\Box}$ provides the guarantee that generating extensions do not contain references to value variables. In order for run-time code generation to be effective, we must ensure that it has code to specialize. For example, if we were to allow the expression $\lambda f.\texttt{code}\ (f\,3)$, the function f would not be available to the generating extension until run time, when it would take the form of a closure. A closure cannot be inlined or specialized to the value 3, so it would essentially have to be lifted into the generated code as a reference, disabling nearly all benefits of performing run-time code generation.

(4) The new language constructs provided by $\lambda^{\Box}$ are extremely simple. There is one new type constructor with one introduction and one elimination form. Also, $\Box$ types and the new constructs appear to be quite orthogonal to other language constructs, including such imperative features as reference cells and exceptions.

Programs in ML$^{\Box}$ express staging concisely and unambiguously, thereby allowing the expression of many optimizations. However, inlining, which may be expressed in some annotation schemes, cannot be expressed by ML$^{\Box}$, because it would require explicit manipulation of code with free value variables, contradicting item 3 above. Further, such manipulation of open code could easily endanger the type-safety of an `eval`-like construct.

2.5 Compiler

In order to demonstrate that our interpretation of `code` expressions as generating extensions results in a realistic system in practice, we have developed a compilation technique from $ML^{\Box}$ to an extension of the Categorical Abstract Machine (CAM), called the CCAM (Code-generating CAM). The primary addition of the CCAM to the CAM is a single parameterized instruction, `emit`, which writes its instruction parameter to a code context which the machine can later execute. As expected, some specializing programs exhibit dramatic reductions in instructions executed on our CCAM simulator when compared with non-specializing versions of the same programs. The CCAM, the compilation strategy, the compiler, and some initial experiments are detailed in Wickline, Lee, and Pfenning [1998].

3. RELATED WORK

Nielson and Nielson's two-level functional language manipulates only "code-closed" programs (i.e. property 3), as $\lambda^{\Box}$ does, but does not have any equivalent of an `eval` operator, and cannot express programs with more than two stages. Davies and Pfenning [1996] details a conservative embedding of the two-level functional language in $ML^{\Box}$. Davies [1996] describes the language $\lambda^{\bigcirc}$, which has annotations from linear-time temporal logic. This language can have any number of stages and does not manipulate exclusively closed code, allowing it to describe inlining effectively. Unfortunately it supports no safe `eval` operator. MetaML [Taha and Sheard 1997] contains an `eval` operator and is able to express inlining in much the same way as $\lambda^{\bigcirc}$. However, Taha and Sheard's notion of "cross-stage persistence" allows variables in code expressions which are not bound to code, which means that MetaML does not support property 3.

4. CONCLUSION

Typed languages can be a useful tool for staging analysis. We have investigated the use of the modal language $ML^{\Box}$ to stage programs for run-time code generation. We argue that this language provides a simple, flexible, and sound framework in which to describe run-time code generation, where the programmer can be assured that the specialization that they desire will be performed.

REFERENCES

Davies, R. 1996. A temporal-logic approach to binding-time analysis. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science* (New Brunswick, New Jersey, 27–30 July 1996), pp. 184–195. IEEE Computer Society Press.

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. 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.

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.

Taha, W. and Sheard, T. 1997. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997* (1997), pp. 203–217.

Wickline, P., Lee, P., and Pfenning, F. 1998. Run-time code generation and Modal ML. Technical Report CMU-CS-98-100 (January), School of Computer Science, Carnegie Mellon University. Also appears as CMU-CS-FOX-98-01.