Run-time Code Generation and Modal-ML *

Philip Wickline Peter Lee Frank Pfenning

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213–3891
{philipw,petel,fp}@cs.cmu.edu

Abstract

This paper presents a typed programming language and compiler for run-time code generation. The language, called $\mathrm{ML}^{\Box}$, extends ML with modal operators in the style of the Mini-$\mathrm{ML}_e^{\Box}$ language of Davies and Pfenning. $\mathrm{ML}^{\Box}$ allows programmers to use types to specify precisely the stages of computation in a program. The types also guide the compiler in generating target code that exploits the staging information through the use of run-time code generation. The target machine is currently a version of the Categorical Abstract Machine, called the CCAM, which we have extended with facilities for run-time code generation.

This approach allows the programmer to express the staging that he wants directly to the compiler. It also provides a typed framework in which to verify the correctness of his staging intentions, and to discuss his staging decisions with other programmers. Finally, it supports in a natural way multiple stages of run-time specialization, so that dynamically generated code can be used in the generation of yet further specialized code.

This paper presents an overview of the language, with several examples of programs that illustrate key concepts and programming techniques. Then, it discusses the CCAM and the compilation of $\mathrm{ML}^{\Box}$ programs into CCAM code. Finally, the results of some experiments are shown, to demonstrate the benefits of this style of run-time code generation for some applications.

1 Introduction

A well-known technique for improving the performance of a computer program is to separate its computations into distinct stages. If this is done carefully, the results of early computations can be exploited by later computations in a way that leads to faster execution (Pike, Locanthi, and Reiser 1985; Jørring and Scherlis 1986; Massalin and Pu 1989). In recent years, a number of researchers have shown that compilers, if made knowledgeable about staged computation, can compile staged programs into target code that performs run-time code generation (Keppel, Eggers, and Henry 1993; Consel and Noël 1996; Engler, Hsieh, and Kaashoek 1996; Auslander, Philipose, Chambers, Eggers, and Bershad 1996; Lee and Leone 1996; Grant, Mock, Philipose, Chambers, and Eggers 1997), sometimes reaping substantial performance gains in the process. The basic advantage of run-time code generation is that it allows low-level code optimizations that are difficult to express as source-to-source transformations, such as register allocation and instruction selection, to take advantage of values that are not known until run time. Dynamic information also permits some optimizations such as loop unrolling and array-bounds checking removal to be pursued more aggressively than is safe at compile time.

In order to make use of run-time code generation, a compiler must first understand how the program's computations are staged. Determining this staging information is not a simple matter, however. While automatic binding-time analyses have been used by partial evaluators and some compilers, we are interested here in developing a programming language that supports a principled and systematic method for describing computational stages and checking their consistency. For this purpose, we have developed $\mathrm{ML}^{\Box}$, an extension of the Mini-$\mathrm{ML}_e^{\Box}$ language of Davies and Pfenning (1996) to nearly all core ML constructs. This language provides modal operators that allow a programmer to specify precisely, through types, the stages of computation in a program. Besides providing the programmer with full control over when and where run-time code generation occurs, the overall implementation is relatively simple, since the complexity of a sophisticated automatic analysis is replaced by an ML-style type checker. Furthermore, by taking a principled approach to program staging, we can work in a framework that facilitates formal development, including precise definitions of the language and its implementation.

In order to demonstrate these advantages, we have implemented a prototype compiler for our language. The compiler generates code for a version of the Categorical Abstract Machine (Cousineau, Curien, and Mauny 1987), called the CCAM, which is extended with a facility for emitting fresh code at run time. An interesting technical issue arises in the compilation process. $\mathrm{ML}^{\Box}$ programs, which may describe programs which specialize in any number of stages, must

*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.

be compiled to a machine which can directly encode only one level of specialization. We have chosen to restrict the CCAM in this way in order to avoid the code-size blowup that can occur when generating machine instructions that generate machine instructions. Resolving this mismatch between the n-level language and the 2-level machine is one of the chief technical contributions of this paper. Our compilation technique is formalized and presented carefully in the hope that the ideas embodied in it may be useful in the development of other multi-staged run-time code generating systems, regardless of the source languages they use.

We begin the paper by explaining some of the motivations for staged computation, and follow this in Section 3 with a brief introduction to the basic constructs of $\mathrm{ML}^{\Box}$. Then, in Section 4, we give a series of programming examples, to show what it is like to write staged programs in our language. These examples are chosen to illustrate different aspects of staged computation, including multi-stage specialization. Section 5 presents the CCAM, our abstract machine for run-time code generation. This is followed in Section 6 by a thorough description of how $\mathrm{ML}^{\Box}$ programs are compiled into CCAM code. Section 7 presents reduction counts on the CCAM for staged and unstaged versions of some of the example programs in Section 4. We end with a brief discussion of related work.

2 Program Staging

The temporal separation of the computations of a program, or "staging," is usually done manually. A notable exception to this is partial evaluation (Jones, Gomard, and Sestoft 1993), in which the staging of programs is performed semi-automatically, according to a programmer-supplied indication of which program inputs will be available in the first stage of computation. This information is used to synthesize a *generating extension* that will generate specialized code for the late stages of the computation when given the first-stage inputs. Both the Tempo system (Consel and Noël 1996) and DyC (Grant, Mock, Philipose, Chambers, and Eggers 1997) use forms of binding-time analysis for the automatic staging of C programs. While much work on partial evaluation has focused on programs with two stages, more recent work has extended the partial evaluation framework to account for multiple computation stages (Glück and Jørgensen 1995).

More common is the use of a programming notation or annotation scheme to support programmer-specified staging. The backquote and antiquote notation of Lisp macros, for example, provides an intuitive though highly error-prone approach to staged computation. A particularly nasty error is the inadvertent capture of variables, leading to the evaluation of variables in incorrect contexts. This has motivated relatively complex "hygienic" macro systems (Kohlbecker, Friedman, Felleisen, and Duba 1986). Problems involving infinite expansion of macros are also quite common in practice. More recently, this notion of backquote/antiquote has been introduced as an extension to the C programming language, in the language 'C (Engler, Hsieh, and Kaashoek 1996). While 'C has an advantage over Lisp in providing a degree of static type checking, the fundamental problems of variable capture and infinite expansion remain.

The Fabius system (Lee and Leone 1996) uses another approach to programmer-specified staging. In Fabius, the staging of ML programs is specified by currying the arguments to functions—a function of type $\tau_1 \to \tau_2 \to \tau_3$, when applied to an argument of type τ_1, causes the dynamic generation of optimized code for the resulting function of type $\tau_2 \to \tau_3$. While this scheme has the advantage that all Fabius programs are also legal ML programs, it is also quite limited in its ability to express staging decisions.

We claim that $\mathrm{ML}^{\Box}$ can be used as a clear and expressive notation for staged computation. Drawing on previous work on the language Mini-$\mathrm{ML}_e^{\Box}$ (Davies and Pfenning 1996), and on the interpretation of this language for run-time code generation described in Wickline, Lee, Pfenning, and Davies (1998), we present an implementation of a prototype compiler for a version of the ML language that uses modal operators to specify precisely the stages of a program's computation. We believe that using the modal source language has the following advantages:

- The programmer is able to express the staging that he wants to the compiler directly, rather than through a heavyweight (and, in practice, unpredictable) analysis.
- The programmer is given a framework which allows him to verify the correctness of his staging intentions. A staging error becomes a type error which can be analyzed and fixed, rather than simply resulting in a slow or incorrect program. Furthermore, this framework is useful for conceptualizing and discussing staging with other programmers through typing specifications.
- This approach is complementary to the use of automatic staging through binding-time analysis. A compiler is free to augment the staging requirements from a hand-staged program using other means.
- The language naturally handles situations in which more than two stages are desired, such as Fabius-style multi-stage specialization (Leone and Lee 1998). This arises, for example, when dynamically generated code is used to compute values that are used in the specialization of yet more code.

3 Modal ML

We briefly introduce the language $\mathrm{ML}^{\Box}$, our extension of Mini-$\mathrm{ML}_e^{\Box}$ (Davies and Pfenning 1996). While we present only the most basic and important operators of $\mathrm{ML}^{\Box}$ here because of space considerations, the compilation technique described in Section 6 extends easily to all core ML constructs. Indeed, our prototype compiler for $\mathrm{ML}^{\Box}$ handles almost all of core ML.

3.1 Syntax

$\mathrm{ML}^{\Box}$ arises from the simply-typed λ-calculus by adding a new type constructor $\Box$. Except for the addition of a new primitive operator, `lift`, it is related to the modal logic S_4 by an extension of the Curry-Howard isomorphism, where $\Box\tau$ means "τ is necessarily true".

In our context, we think of $\Box\tau$ as the type of *generators for code of type* τ. Generators are created with the construct, `code` M, where M is any $\mathrm{ML}^{\Box}$ expression. For example, $\vdash$ `code` $(\lambda x.x) : \Box(\alpha \to \alpha)$ is a generator which, when invoked, generates code for the identity function. Figure 1 presents the syntax of $\mathrm{ML}^{\Box}$ which renames the `box` and `let box` constructs in (Davies and Pfenning 1996) to `code` and `let cogen`, respectively. Note that there are two kinds of variables: *value variables* bound by λ (denoted by x) and *code variables* bound by `let cogen` (denoted by u). We use a to range over either type of variable.

Type Variables α, β, γ			
Types	τ, σ	::=	$\alpha \mid \tau \rightarrow \sigma \mid \Box\tau$
Terms	M, N	::=	$x \mid \lambda x.M \mid MN \mid u \mid$ `code` $M \mid$ `lift` $M \mid$ `let cogen` $u = M$ `in` N `end`
Contexts	Γ	::=	$\cdot \mid \Gamma, x : \tau \mid \Gamma, u : \tau$

Figure 1: Basic ML$^\Box$ Syntax

To invoke a generator, one might expect a corresponding `eval` construct of type $(\Box\alpha) \rightarrow \alpha$. Such a function is in fact definable in ML$^\Box$, but is not a part of the basis of the language. Instead we have a binding construct `let cogen` $u = M$ `in` N `end` which expects a code generator M of type $\Box\tau$ and binds a code variable u. However, even evaluation of `let cogen` $u = M$ `in` N `end` will *not* immediately generate code. Instead, the generation of specialized code for M is deferred until the code variable u is encountered during evaluation. For example,

$$\vdash \lambda x.\texttt{let cogen}\ u = x\ \texttt{in}\ u\ \texttt{end} : (\Box\alpha) \rightarrow \alpha$$

is the function `eval` mentioned above which invokes a code generator and then evaluates that code.

Generation of code is postponed as long as possible so that the context into which the code is emitted can be used for optimizations. For example, the following is a higher-order function which takes generators for two functions and creates a generator for their composition. The result may be significantly more efficient than generating first and then composing the resulting functions. Note that this function returns a generator, but does not call the given generators or emit code itself.

$$\begin{aligned}\vdash \lambda f.\lambda g.&\texttt{let cogen}\ f' = f\ \texttt{in}\\ &\quad\texttt{let cogen}\ g' = g\ \texttt{in}\\ &\quad\quad\texttt{code}\ \lambda x.f'(g'(x))\ \texttt{end end}\\ &: \Box(\beta \rightarrow \gamma) \rightarrow \Box(\alpha \rightarrow \beta) \rightarrow \Box(\alpha \rightarrow \gamma)\end{aligned}$$

Readers familiar with Davies and Pfenning's Mini-ML$_e^\Box$ will notice that we have added a special operator called `lift`, which obeys the rule that `lift` M has type $\Box\tau$ if M has type τ. `lift` M evaluates M and returns a generator which just "quotes" the resulting value. In contrast to the `code` construct, this prohibits all optimizations during code generation. As noted in Davies and Pfenning (1996), `lift` is definable in Mini-ML$_e^\Box$ for base types, but its general form has no basis in the underlying modal logic. Here we show that it nonetheless has a reasonable and useful operational interpretation in the context of run-time code generation.

3.2 Typing Rules

The typing judgment $\Delta; \Gamma \vdash M : \tau$ uses two contexts: a code context Δ in which code variables are declared, and an ordinary context Γ declaring value variables. The typing rules in Figure 2 are the familiar ones for the λ-calculus plus the rules for `let cogen`, `code` and `lift`.

The critical restriction which guarantees proper staging is that only code variables (which occur in Δ) are permitted to occur free in generators (underneath the `code` constructor), but no value variables — this is enforced by the empty value context in the premise of the `code` rule. The `let cogen` rule expresses that if we have a value which is a code generator (and therefore of type $\Box\tau$), we can bind a code variable u of type τ which may be included in other code generators.

4 Programming with ML$^\Box$

In order to give a feeling for what it is like to write ML$^\Box$ programs, we present several examples.

4.1 Computing the Value of Polynomials

To start with a simple example, consider the following ML function which evaluates a 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 = [2,4,0,2333];

(* 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. 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. Furthermore, code-level optimizations cannot take advantage of the staging, for example in instruction selection and register allocation, as described by Lee and Leone (1996). Therefore we rewrite `specPoly` as the ML$^\Box$ function `compPoly`.

```
(* 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 codeGenerator = compPoly poly1;
val mlPolyFun = eval codeGenerator;
```

Here the `code` operator marks the introduction of a code generator, and $[\tau]$ is concrete syntax for $\Box\tau$. 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.

4.2 Libraries

One possibility afforded by ML$^\Box$ is to install staged versions of library routines, so that client applications can benefit from dynamic specialization of the library code.

Consider, for example, placing the `compPoly` function in a library. Then, suppose we have a client program:

$$\frac{x:\tau \text{ in } \Gamma}{\Delta;\Gamma \vdash x:\tau} \qquad \frac{\Delta;\Gamma \vdash M:\tau \to \sigma \quad \Delta;\Gamma \vdash N:\tau}{\Delta;\Gamma \vdash MN:\sigma} \qquad \frac{\Delta;\cdot \vdash M:\tau}{\Delta;\Gamma \vdash \texttt{code}\ M:\Box\tau}$$

$$\frac{\Delta;(\Gamma,x:\tau) \vdash M:\sigma}{\Delta;\Gamma \vdash \lambda x.M:\tau \to \sigma} \qquad \frac{u:\tau \text{ in } \Delta}{\Delta;\Gamma \vdash u:\tau} \qquad \frac{\Delta;\Gamma \vdash M:\tau}{\Delta;\Gamma \vdash \texttt{lift}\ M:\Box\tau} \qquad \frac{\Delta;\Gamma \vdash M:\Box\tau \quad (\Delta,u:\tau);\Gamma \vdash N:\sigma}{\Delta;\Gamma \vdash \texttt{let cogen}\ u = M\ \texttt{in}\ N\ \texttt{end}:\sigma}$$

Figure 2: Typing rules for $\mathrm{ML}^{\Box}$

```
(* val client : t1 -> [t2 -> t3] *)
fun client x =
  ... code (fn y => ... compPoly (makePoly y) ...)
  ...
```

where `makePoly :  t2 -> poly`. Even though the `client` program does not have access to the source code of the `compPoly` library routine, it is still able to benefit from the fact that it will perform run-time code generation on the polynomial computed by `makePoly`.

This example also illustrates one way that multi-stage specialization can be achieved in our system. Note that the `client` program takes the argument `x` and generates code for a function of type `t2 -> t3`, and that it is this dynamically generated code that invokes the `compPoly` function. Hence, dynamically generated code can compute values which in turn are used to generate yet more code. This kind of multi-stage specialization is difficult to achieve in standard partial evaluation, but is supported naturally in our framework.

4.3 Packet Filters

A packet filter is a procedure invoked by an operating system kernel to select network packets for delivery to a user-level process. To avoid the overhead of a context switch on every packet, a packet filter must be kernel resident. But kernel residence has a distinct disadvantage: it can be difficult for a user-level process to specify precisely the types of packets it wishes to receive, because packet selection criteria are different for each application and can be quite complicated. A commonly adopted solution to this problem is to allow user-level processes to install a program that implements a selection predicate into the kernel's address space (McCanne and Jacobson 1993). In order to ensure that the selection predicate will not corrupt internal kernel structures, the predicate must be expressed in a "safe" programming language. Unfortunately, this approach has a substantial overhead, since the safe programming language is typically implemented by a simple (and therefore easy-to-trust) interpreter.

As demonstrated by several researchers, run-time code generation can eliminate the overhead of interpretation by specializing the interpreter to each packet filter program as it is installed. This has the effect of compiling each packet filter (Massalin and Pu 1989; Engler, Wallach, and Kaashoek 1995; Lee and Leone 1996; Sirer, Savage, Pardyak, DeFouw, and Bershad 1996). To demonstrate this idea in our language, consider the following excerpt of the implementation of a simple interpreter for the BSD packet filter language (McCanne and Jacobson 1993) in ML.

```
(* val evalpf : instruction array *
 *            int array *
 *            int * int * int -> int
 * Return 1 to select packet, 0 to reject,
 * ~1 if error *)
fun evalpf (filter, pkt, A, X, pc) =
    if pc > length filter then ~1
    else case sub (filter, pc) of
          RET_A => A
        | RET_K(k) => k
        | LD_IND(i) =>
            let val k = X + i in
              if k > length pkt then ~1
              else evalpf
                  (filter, pkt, sub(pkt,k),
                   X, pc+1)
            end
        ...
```

The interpreter is given by a simple function called `evalpf`, which is parameterized by the filter program, a network packet, and variables that encode the machine state. The machine state includes an accumulator, a scratch register, and program counter.

In order to stage this function, it is straightforward to transform the code so that the packet filter program and program counter are "early" values, and the packet, accumulator, and scratch register are "late." Then, the computations that depend only on the late values can be generated dynamically by enclosing them in `code` constructors. At first glance it might seem odd that the interpreter can be specialized not only to the packet filter program but also the program counter. This is possible since (Berkeley-style) packet filters have no loops and thus always terminate. Thus, specializing the interpreter on the program counter has the effect of "unrolling" the interpreter over all of the instructions in the given packet filter program.

```
(* val bevalpf :
 *      (instruction array * int) ->
 *      [int * int * int array -> int] *)
fun bevalpf (filter, pc) =
    if pc > length filter then (fn _ => ~1)
    else case sub (filter, pc) of
          RET_A => code (fn (A,X,pkt) => A)
        | RET_K(k) =>
            let cogen k' = lift k
            in code (fn _ => k') end
        | LD_IND(i) =>
            let cogen ev = bevalpf (filter, pc+1)
                cogen i' = lift i
            in code (fn (A,X,pkt) =>
                     let val k = X + i' in
                      if k >= length pkt then ~1
                      else ev (sub(pkt,k), X, pkt)
                     end)
        ...
```

When applied to a filter program and program counter, the result of `bevalpf` is the CCAM code of a function that takes a machine state and packet, and computes the result of the packet filter on that packet and state. Later, in Section 7, we show that the improvement in execution time for a typical BPF packet filter is substantial.

4.4 Memoizing ML$^\Box$ Programs

Since specializing programs at run time typically involves additional expense, a central assumption of this approach is that the specialized code generated will often be used many times. This happens naturally in some programs. If, for example, a program specializes a section of code and then immediately, in the same scope in the code, uses that specialized code many times, it is easy to bind the generated code to a variable and use that variable, thereby avoiding regeneration of the code. In other situations we must work harder to get this sort of "memoizing" behavior.

Consider the following specializing function to compute the value of an integer raised to the power of e.

```
(* val codePower : int -> [int -> int] *)
fun codePower e =
    if e = 0 then code (fn _ => 1)
    else let cogen p = codePower (e - 1)
        in code (fn b => b * (p b)) end
```

If this function is used to compute powers in two or more sections of the same program, it is possible that the same code will be generated and regenerated many times, making the resulting program *slower* rather than faster. We must carefully arrange to have generated programs saved for future use in situations where we they are likely to be needed again. Fortunately, we can bind up this functionality with the function itself. We assume an implementation of tables and table `specCode` with the types given below.

```
(* specCode : (int -> int) table
   get : 'a table * int -> 'a option
   add : 'a table * (int * 'a) -> unit *)

(* memoPower1 : int -> int -> int *)
fun memoPower1 e =
    case lookup (specCode, e) of
        NONE => let cogen p = codePower e
                    val p' = p
                in add (specCode, (e, p')); p' end
      | SOME p => p;
```

This function simply embeds the `codePower` function within a wrapper that checks a table to determine whether or not a particular specialized version of the function exists. If it does, then it is returned, without need for further work. Otherwise, `codePower` is called, and a new function is generated, stored in the table, and returned.

While `memoPower1` saves generated code, so that it will benefit from past computations on the *same* exponent, it does nothing to speed up the computation for two different exponents, even though they may share subcomputations.

`memoPower2` goes even further than `memoPower1`. It saves the result of each internal call to the power function in a table, `genExts`, of generating extensions. Then if it is called to compute, for instance, n^{65} and then m^{34} it will not have to do any additional work to make a generating extension for the second call.

```
(* specCode : (int -> int) table
   genExts : [int -> int] table
   get : 'a table * int -> 'a option
   add : 'a table * (int * 'a) -> unit *)

(* memoPower2 : int -> int -> int *)
fun memoPower2 e =
    (case lookup (specCode, e) of
         NONE =>
         let cogen p = mPower e
             val p' = p
         in add (specCode, (e, p')); p' end
       | SOME p => p)

(* mPower : int -> [int -> int] *)
and mPower e =
    (case lookup (genExts, e) of
         NONE => let val p = bPower e
                 in add (genExts, (e, p)); p end
       | SOME p => p)

(* mPower : int -> [int -> int] *)
and bPower e =
    if e = 0 then code (fn _ => 1)
    else let cogen p = mPower (e - 1)
         in code (fn b => b * (p b)) end;
```

While specifying memoization behavior by hand in this fashion may be tedious in some cases, it does allow the programmer to control carefully what and how memoization will occur. Furthermore, generic memoization routines can accommodate most common memoization needs.

5 The CCAM

In this section we present the CCAM, an *ad hoc* extension of the CAM (Cousineau, Curien, and Mauny 1987) which provides facilities for run-time code generation and is the target of the compiler detailed in the next section.

For reasons explained below, we would like to model the form of run-time code generation provided by the Fabius compiler (Lee and Leone 1996), which does not manipulate source terms at run time. The design of an abstract machine for this kind of code generation must strike a delicate balance. On one hand, we want to abstract away from the details of individual architectures as much as possible. On the other hand, a realistic compiler must directly generate machine code at run time, which by its very nature depends on the specific machine architecture. The CCAM therefore has instructions to emit instructions directly into a code block, but we carefully limit the power of such *emit* instructions so they can serve as a generic model from which compilation schemes for individual machine architectures can be derived.

5.1 Run-Time Code Generation

Many techniques have been used to build systems which dynamically specialize programs at run time. The most naïve technique is to perform, at run time, source level substitution on the program to be specialized and then call a full compiler. Obviously, this technique is extremely expensive. Less expensive schemes have been developed for run-time specialization, notably template filling, which is used by the Tempo system (Consel and Noël 1996) and the Synthesis kernel (Massalin and Pu 1989). Template filling compiles selected parts of the program to sequences of machine code with "holes", that is, sections whose instructions need to be filled in at run time. These templates may then be copied and have their holes plugged to produce specialized machine code. Template filling is typically much more efficient than calling a full compiler, because source-level terms are never manipulated at run time, and most of the compilation (all of the template that is not a hole) can be computed ahead of time. However, template filling is inflexible about the kinds of values which may fill its holes and the kinds of optimizations that may be performed at run time.

We intend our abstract machine to model run-time code generation. In particular, there are three important features that we believe allow substantial flexibility of specialization at relatively low cost.

- The machine should not manipulate source-level terms at run time. Instead machine language programs should be synthesized directly from machine language programs.
- The machine should encode terms to be specialized directly into the instruction stream, for example in the form of immediate operands to instructions. This is in contrast to systems which copy templates and fill in holes at run time. Instruction stream encoding enables a great deal of flexibility in the kinds of specialization that can be performed at run time.
- The machine must allow dynamic staging of code, i.e., the number of stages of program specialization may depend on some value that will not be known until run time. This is necessary to fully exploit the specialization opportunities in many situations. For example, many programs have a top-level loop which waits to receive input in some form, and then takes appropriate action. In contrast, conventional off-line partial evaluation will fail to serve such a program well because even multi-level partial evaluation has no way to specialize on each of the variably many inputs.

5.2 The CAM

The CAM (Cousineau, Curien, and Mauny 1987) is a simple abstract machine which uses categorical combinators as its instructions. The CAM allows a great deal of flexibility in manipulating environments, which are represented as nested pairs of values. This flexibility makes the CAM an especially good choice as a base for our code-generating abstract machine. For the purposes of this paper, the reader need not have knowledge of categorical logic. We use the CAM only as a simple, stack-based machine.

5.2.1 Instructions

The syntactic definitions of various CAM components follow.

Instructions	i	$::=$	$\mathtt{id} \mid \mathtt{fst} \mid \mathtt{snd} \mid \mathtt{push} \mid \mathtt{swap} \mid \mathtt{cons} \mid \mathtt{app} \mid \text{‘}e \mid \mathtt{Cur}(P)$
Values	e, f	$::=$	$() \mid [e : P] \mid (e, f)$
Programs	P	$::=$	$i; \cdots ; i$
Stacks	S	$::=$	$\cdot \mid e :: S$

We will also use ‘$\cdot$’ to represent the empty program, with the understanding that $(\cdot; P) = (P; \cdot) = P$.

The values of the CAM consist of the empty tuple, $()$; closures, created when `Cur` instructions execute; and pairs, which also serve to represent the environment. The environment consisting of values $f_1, \ldots, f_n$ is represented as the value $(((), f_1), \ldots, f_n)$.

5.2.2 Transitions

A state of the CAM consists of a stack paired with an instruction sequence, (S, P). We think of the top element e of the stack ($e :: S$) as the current environment in which the program P is to be executed. It is characteristic of the CAM that the value returned by an instruction is also placed on top of the stack. The upper part of Figure 3 describes the transitions of the CAM. We write $(S, P) \Longrightarrow (S', P')$ if the stack S and program P match the first two columns of Figure 3 while S' and P' match the last two. $\Longrightarrow^*$ denotes the reflexive transitive closure of the $\Longrightarrow$ relation.

The instruction `id` leaves the stack unchanged. `fst` and `snd` project the first and second elements of pairs found on top of the stack, respectively. A variable which accesses the current environment (on top of the stack) is therefore compiled to a sequence of `fst` instructions followed by a `snd` instruction. The instructions `push`, `swap`, and `cons` manipulate the stack, and taken as a trio constitute the pairing mechanism of the machine. If $\mathtt{push}; P_1; \mathtt{swap}; P_2; \mathtt{cons}$ is run with the environment e on top of the stack, then `push` will save e by making another copy on the stack. P_1 will then consume the first copy of e, replacing it with its return value f_1. The `swap` instruction will restore the second copy of e to the top of the stack, saving f_1 in the process. P_2 will then consume e and replace it with its return value f_2. Finally, `cons` will pair the top two elements of the stack, f_1 and f_2, making the pair (f_1, f_2) the return value for the whole program. To emphasize the pairing operation of these instructions, we will often write ‘$\langle$’ for `push`, ‘`,`’ for `swap`, and ‘$\rangle$’ for `cons`, and drop the separating semicolons. Thus $\mathtt{push}; P_1; \mathtt{swap}; P_2; \mathtt{cons}$ may be written as $\langle P_1 , P_2 \rangle$.

$\mathtt{Cur}(P')$ creates a closure $[e : P']$ from the current environment e on top of the stack and P'. The corresponding `app` instruction expects a pair consisting of a closure $[e : P']$ and its argument f on top of the stack. It applies the function by evaluating its body P' in the environment e extended by f, represented by (e, f).

5.3 An Abstract Machine for Run-time Code Generation

The CCAM has been designed as an extension of the CAM with the goals listed in Section 5.1 in mind. The primary novelty of the CCAM is the $\mathtt{emit}(i)$ instruction. It represents the series of instructions required on a real computer to produce the instruction i in a specialized program. As will be made more clear below, the CCAM encodes a generating extension as a series of $\mathtt{emit}(i)$ instructions.

As an example of this form of code generation, consider the instruction `emit(add)`. If this instruction were compiled to real machine instructions it might be represented by three instructions, one which contained the lower 16 bits of the add instruction in an immediate load low instruction, one which contained the upper 16 bits, and finally one to write the assembled instruction to memory. A more sophisticated specialization system might compile `emit(add)` to a series of instructions which tests the values of the operands of the add instruction at specialization time (if they are available) and eliminate the instruction altogether if either one is 0.

5.3.1 Nested Code Emission

Multi-staged programs are a potential problem for our abstract machine. It is clearly possible, in $\mathrm{ML}^{\Box}$, to write programs containing expressions such as $\mathtt{code}\ (\cdots \mathtt{code}\ M \cdots)$ in which there are nested code generators. If we encode generating extensions with $\mathtt{emit}(i)$ instructions, must such programs give rise to instructions of the form $\mathtt{emit}(\mathtt{emit}(i))$? If so, then a chain of n generating extensions could lead to n nested emits.

Observe, however, that on a machine with fixed-length instructions, there is a limited amount of space available for immediate operands, and so if instructions to be emitted are embedded in instructions in the instruction stream, it will take at least two instructions to represent one emitted instruction. Furthermore it could take 2^n instructions to represent $\overbrace{\texttt{emit(emit(}\cdots\texttt{emit}}^{n}(i)\cdots))$. For this reason, nested emits are not allowed on the CCAM, and our compilation scheme therefore needs to take special steps in order to allow multi-level specialization.

Note that for a similar reason, emitting a $\texttt{Cur}(P)$ instruction would be unrealistic and is therefore also disallowed.

5.3.2 Instructions

The CCAM has the usual seven instructions associated with the CAM, and five more for code generation. The new syntactic definitions of the CCAM follow.

Simple Inst	i	$::=$	`id` | `fst` | `snd` | `push` | `swap` | `cons` | `app` | $`e$ | `lift` | `arena` | `merge` | `call`
Composite Inst	I	$::=$	i | $\texttt{emit}(i)$ | $\texttt{Cur}(P)$
Values	e, f	$::=$	(e, f) | $[e : P]$ | B | $()$
Code Blocks	B	$::=$	$\{P\}$
Programs	P	$::=$	$I; \cdots ; I$
Stacks	S	$::=$	$\cdot$ | $e :: S$

Instructions are now broken into simple and composite instructions. This division is to enforce the restriction we note above on the arguments of the `emit` instruction.

The CCAM has a new kind of value, code blocks, not available in the CAM. A code block is a dynamically created sequence of instructions. `emit` instructions, when executed, append their arguments to the end of code blocks. On a real computer, a code block would be represented by a pointer to the *end* of a dynamically created segment of machine code. When a new instruction is added to the sequence, the pointer is incremented appropriately. All of the new instructions of the CCAM manipulate code blocks.

5.3.3 Transitions

The CCAM is a conservative extension of the CAM in that it has all of the same values, instructions, and transitions that the CAM has. Figure 3 describes the transitions of the CCAM. There is one new transition for each new instruction:

- `arena` creates an empty code block on top of the stack,
- $\texttt{emit}(i)$, emits instruction i into the current code block at the head of the current environment,
- `lift` directly inserts a quoted value into the current code block,
- `merge` inserts a function into the code block, and
- `call` invokes the current code block by inserting it into the instruction stream.

This use of these instructions will be explained in more detail in Section 6.2.

6 Compilation

In this section we describe our technique for compiling ML$^\square$ terms into CCAM code. While the compilation of normal ML terms to the CAM is straightforward, our task is considerably more difficult. The principal difficulty comes from the fact that ML$^\square$ programs can specify multiple rounds of dynamic code generation so that, for example, the results computed by dynamically generated code can be used to specialize yet more code. This feature of ML$^\square$ is problematic because the CCAM does not have any facility for emitting `emit` instructions. Thus, a primary technical contribution of this paper is the strategy for compiling multi-stage programs down to code appropriate for a two-level machine. It should be noted that this scheme is applicable to languages and machines beyond those considered in this paper. Therefore we provide a detailed explanation of the compilation scheme and its development in the hope that this may serve as a road map to designing similar compilation schemes for other multi-level languages. We begin with a high-level overview of our compilation strategy in Section 6.1 before diving into the rather technical details of compilation to the CCAM in Section 6.2. Some readers, particularly those unfamiliar with the CAM, may prefer to skip Section 6.2 in favor of a more careful reading of Section 6.1.

6.1 Strategy

As a first attempt at a compilation strategy, if we would normally compile the term M to the series of instructions $i_1; i_2; \cdots ; i_n$, we can compile the term $\texttt{code}\ M$ to the instructions $\texttt{emit}(i_1); \texttt{emit}(i_2); \cdots ; \texttt{emit}(i_n)$ to produce a run-time code generator for M. While this is the basic idea of our compilation technique, this simple strategy neglects two important details. First, it does not take account of the possibility of free code variables in an expression and the specialization opportunities presented by those variables. Second, it does not explain how to deal with code generators which are nested within code generators.

In order to benefit from run-time specialization, the code generators need to make use of values that are computed at run time. In our system, this new information is supplied through the free code variables in a `code` expression. These variables become bound to code generators which can be called to substitute code into the containing expression. To make this more clear, consider the following ML$^\square$ term:

$$\texttt{let cogen}\ u = \texttt{code}\ 0\ \texttt{in code}\ \lambda x.u * x\ \texttt{end}$$

In Section 6.2 we describe in some detail the compilation of ML$^\square$ terms into CCAM code. Here, we try to avoid the rather intricate details of the CCAM, and instead simply underline all sub-terms which would be compiled into series of `emit` instructions. For the current example, then, we obtain the following:

$$\texttt{let cogen}\ u = \texttt{code}\ \underline{0}\ \texttt{in code}\ \underline{\lambda x.}u\,\underline{*x}\ \texttt{end}$$

Note that the code variable u in the expression $\texttt{code}\ \underline{\lambda x.}u\,\underline{*x}$ is *not* underlined, that is, it will not be translated into `emit` instructions. Instead, when $\texttt{code}\ \underline{\lambda x.}u\,\underline{*x}$ is activated as a code generator, it will in turn activate the code generator for $\texttt{code}\ \underline{0}$ bound to u. Therefore the whole expression will end up emitting $\lambda x.0 * x$.

Although we do not address in this paper exactly what kinds of low-level optimizations will be performed by our `emit` instructions, it is easy to imagine that specializing code for $\underline{\lambda x.}u\,\underline{*x}$ could test to see if the value being substituted for u is a zero, and if so specialize to $\lambda x.0$. This and other similar optimizations are performed, for example, by the Fabius compiler (Lee and Leone 1996).

We now consider the problem of nested code generators. Naïvely, if M compiles to $i_1; i_2; \cdots ; i_n$, then $\texttt{code}\ M$ is compiled into

$$\texttt{emit}(i_1); \texttt{emit}(i_2); \cdots ; \texttt{emit}(i_n)$$

Stack	*Program*	*Stack*	*Program*
S	$\texttt{id}; P$	S	P
$(e, f) :: S$	$\texttt{fst}; P$	$e :: S$	P
$(e, f) :: S$	$\texttt{snd}; P$	$f :: S$	P
$e :: S$	$`f; P$	$f :: S$	P
$e :: S$	$\texttt{push}; P$	$e :: e :: S$	P
$e :: f :: S$	$\texttt{swap}; P$	$f :: e :: S$	P
$e :: f :: S$	$\texttt{cons}; P$	$(f, e) :: S$	P
$e :: S$	$\texttt{Cur}(P'); P$	$[e : P'] :: S$	P
$([e : P'], f) :: S$	$\texttt{app}; P$	$(e, f) :: S$	$P'; P$
$e :: S$	$\texttt{arena}; P$	$\{\cdot\} :: S$	P
$(e, \{P'\}) :: S$	$\texttt{emit}(i); P$	$(e, \{P'; i\}) :: S$	P
$(e, \{P'\}) :: S$	$\texttt{lift}; P$	$(e, \{P'; `e\}) :: S$	P
$(\{P'\}, (e, \{P''\})) :: S$	$\texttt{merge}; P$	$(e, \{P''; \texttt{Cur}(P')\}) :: S$	P
$(e, \{P'\}) :: S$	$\texttt{call}; P$	$e :: S$	$P'; P$

Figure 3: Transitions of the CCAM

and therefore $\texttt{code code } M$ becomes

$$\texttt{emit}(\texttt{emit}(i_1)); \texttt{emit}(\texttt{emit}(i_2)); \cdots; \texttt{emit}(\texttt{emit}(i_n))$$

Of course, as discussed in Section 5.3.1, such instructions are not allowed on the CCAM. Our solution is to avoid repeated specialization of the inner code generator by lifting it unchanged into the specialized code of the outer code generator. For example, instead of treating $\texttt{code}\ (\cdots \texttt{code}\ M \cdots)$ as the three-level term $\texttt{code}\ \underline{(\cdots \texttt{code}\ \underline{M} \cdots)}$, we can compile it as the two-level term

$$\texttt{let cogen } u = \texttt{lift } (\texttt{code } \underline{M}) \texttt{ in code } (\underline{\cdots u \cdots}) \texttt{ end}$$

Of course, this is not entirely satisfactory, since M may contain variables which are bound in the enclosing $\texttt{code}$ expression. For example, if we transform the expression

$$\texttt{code } (\texttt{let cogen } u' = N \texttt{ in code } u' + M \texttt{ end})$$

to

$$\begin{array}{l}\texttt{let cogen } u = \texttt{lift } (\texttt{code } u' \underline{+M}) \\ \texttt{in code } (\underline{\texttt{let cogen } u' = N \texttt{ in}}\ u\ \underline{\texttt{end}}) \texttt{ end}\end{array}$$

as we did above, then u' is unbound in $\texttt{code } u' \underline{+M}$. For this reason, we need to arrange for the code generator which is lifted into the enclosing generator to use the environment that is active where it was originally located, not where it is lifted into the environment. It is not possible to represent this operation in our source language, but it is quite easy to do so on the CCAM, since no distinction is made between environments and other data values. When translating to the CCAM, a nested code generator is compiled into a function which accepts as its argument an environment, with which it replaces its current environment. This function is then inserted into the specialized code via the `lift` operator. Finally, code is emitted that applies the lifted function to the environment that will be available when the specialized code is run.

The net result of our compilation scheme is that it yields run-time code generators which *do not* incrementally specialize code, even if that code is nested under several `code` constructs. Instead code is generated in a lazy fashion, only when it is needed and only when all of the code variables in it are bound. The primary advantage of this, as we have noted, is that nested `emit` instructions are not necessary, thereby avoiding possible exponential code blowup. Another advantage is that if intermediate stages of code generators in a multiply nested code generator will not be used more than once, this strategy will be faster than an incrementally specializing strategy, without sacrificing the quality of the specialized code which is the end product. On the other hand, if the intermediate stages will be used repeatedly, an incremental strategy may be preferable.

6.2 Details of the Compilation Scheme

We use variable contexts Ω and Λ to keep track of the position of bound variables in the run-time environment, so that correct code can be generated for variable references. They are formed as follows.

$$\text{Variable Contexts} \quad \Omega, \Lambda \quad ::= \quad () \mid (\Omega, a)$$

We will define a pair of functions, $[\![M]\!]_\Omega$ and $[\![M]\!]^\Lambda_\Omega$, from $\mathrm{ML}^\Box$ terms to CCAM programs. The first, $[\![M]\!]_\Omega$, compiles an $\mathrm{ML}^\Box$ term M with free variables in Ω into normal CCAM code. The second, $[\![M]\!]^\Lambda_\Omega$, translates M with free variables either in Ω or Λ into a generating extension for M, specializing it to the variables contained in Λ.

6.2.1 Preliminaries

In the compilation rules, we will apply contexts to variables as if they were functions. This is intended to represent the CCAM code necessary to select the argument variable from the context. For example, if Ω is the context $(\cdots((\Lambda, a_n), a_{n-1}), \cdots, a_0)$ then $\Omega(a_n) = \underbrace{\texttt{fst}; \cdots; \texttt{fst}}_{n}; \texttt{snd}$.

We also need a context extension operation $\Lambda|\Omega$. If Λ and Ω are variable contexts, and $\Omega = (\cdots(((), a_1), a_2), \cdots, a_n)$ then we write $\Lambda|\Omega$ to represent the extended context of form $(\cdots((\Lambda, a_1), a_2), \cdots, a_n)$

For brevity and clarity, we will often use underlining to denote the emission of code, so that $\texttt{emit}(i)$ is written as $\underline{i}$. This also applies to our convention of applying contexts as if they were functions; for example, if Ω is the context $(\cdots((\Lambda, a_n), a_{n-1}), \cdots, a_0)$ then $\underline{\Omega(a_n)} = \underline{\texttt{fst}}; \cdots; \underline{\texttt{fst}}; \underline{\texttt{snd}}$.

We now describe the compilation of $\mathrm{ML}^\Box$ terms into CCAM code. $\mathrm{ML}^\Box$ is a a full-featured dialect of ML that includes datatypes, pattern-matching, and references. Since space does not permit us to describe the compilation of all of the features of ML$^\square$, we will focus on the core features that pertain most directly to staged computation.

6.2.2 Compiling core ML terms

The compilation function $[\![M]\!]_\Omega$ is used to compile ML$^\square$ terms M which are not enclosed in a `code` construct in the context Ω. At run time, the term M will be evaluated in an environment which matches Ω, supplying values for its value variables and code generators for its code variables.

For the base ML constructs, compilation is done in exactly the same way as described for the CAM (Cousineau, Curien, and Mauny 1987). The fundamental invariant to keep in mind is that if the value of M in environment e is f, then on the CAM

$$(e :: S\ ,\ [\![M]\!]_\Omega; P) \Longrightarrow^* (f :: S\ ,\ P).$$

In other words, the instructions for M consume the current environment e on the top of the stack and replace it by the value of M.

Value variable references are compiled to selections from the environment.

$$[\![x]\!]_\Omega = \Omega(x)$$

An abstraction is directly compiled to a `Cur` instruction which will immediately create the appropriate closure value when executed.

$$[\![\lambda x.M]\!]_\Omega = \mathtt{Cur}([\![M]\!]_{(\Omega,x)})$$

On the CAM, application of functions is achieved via the `app` instruction. This instruction expects that the top of the stack will be a pair of a closure and its argument

$$[\![MN]\!]_\Omega = \langle [\![M]\!]_\Omega \mathtt{,} [\![N]\!]_\Omega \rangle; \mathtt{app}$$

Assuming $[e' : P']$ is the value of M and f is the value of N, the evaluation of $[\![MN]\!]_\Omega$ proceeds as follows:

$$\begin{array}{lll}
(e :: S, & \langle [\![M]\!]_\Omega \mathtt{,} [\![N]\!]_\Omega \rangle; \mathtt{app}; P) & \Longrightarrow \\
(e :: e :: S, & [\![M]\!]_\Omega \mathtt{,} [\![N]\!]_\Omega \rangle; \mathtt{app}; P) & \Longrightarrow^* \\
([e' : P'] :: e :: S, & \mathtt{,} [\![N]\!]_\Omega \rangle; \mathtt{app}; P) & \Longrightarrow \\
(e :: [e' : P'] :: S, & [\![N]\!]_\Omega \rangle; \mathtt{app}; P) & \Longrightarrow^* \\
(f :: [e' : P'] :: S, & \rangle; \mathtt{app}; P) & \Longrightarrow \\
(([e' : P'], f) :: S, & \mathtt{app}; P) & \Longrightarrow \\
((e', f) :: S, & P'; P) &
\end{array}$$

Here it helps to remember that '$\langle$' is `push`, '`,`' is `swap` and '$\rangle$' is `cons`.

6.2.3 Compiling code generators

While `let cogen` $u = M$ `in` N `end` is treated specially when typing ML$^\square$ terms, operationally it introduces nothing new. It binds the result of evaluating M to u in the environment. The following sequence accomplishes precisely this.

$$[\![\mathtt{let\ cogen}\ u = M\ \mathtt{in}\ N\ \mathtt{end}]\!]_\Omega = \langle [\![M]\!]_\Omega \rangle; [\![N]\!]_{(\Omega,u)}$$

Our plan is to compile `code` expressions as described in Section 6.1. Since our code generators must be supplied with a code block into which to emit code, we enclose each generating extension in a `Cur` instruction so that when it is evaluated it will form a closure which can be bound into the environment. We will need to arrange for these closures to be applied to code blocks. The compilation function for generating extensions, $[\![M]\!]_\Omega^\Lambda$, employs two contexts, Λ to hold the free code variables to which the term should be specialized, and Ω to keep track of the local binders under which we have descended. The rule, then, for compiling `code` expressions is surprisingly simple:

$$[\![\mathtt{code}\ M]\!]_\Omega = \mathtt{Cur}([\![M]\!]_{()}^\Omega)$$

Ω contains all of the variables free in M, and in particular the code variables to which we want to specialize M.

The rule for compiling code variables must select out of the environment the code generator to which the variable is bound, create a new code block (accomplished with the `arena` instruction), and apply the code generator to that new code block. Finally, we need to jump (using the `call` instruction) to the code that is deposited in the code block by the generator.

$$[\![u]\!]_\Omega = \langle \Omega(u) \mathtt{,arena} \rangle; \mathtt{app}; \mathtt{call}$$

For references to (non-code) value variables, function applications, and `let cogen` expressions, the compilation function for code generators looks just as one would expect, given the strategy described above.

$$\begin{array}{lcl}
[\![x]\!]_\Omega^\Lambda & = & \underline{\Omega(x)} \\
[\![MN]\!]_\Omega^\Lambda & = & \underline{\langle}[\![M]\!]_\Omega^\Lambda \underline{\mathtt{,}} [\![N]\!]_\Omega^\Lambda \underline{\rangle}; \underline{\mathtt{app}} \\
[\![\mathtt{let\ cogen}\ u = M\ \mathtt{in}\ N\ \mathtt{end}]\!]_\Omega^\Lambda & = & \underline{\langle}[\![M]\!]_\Omega^\Lambda \underline{\rangle}; [\![N]\!]_{(\Omega,u)}^\Lambda
\end{array}$$

These rules are exactly the same as their normal compilation counterparts in $[\![M]\!]_\Omega$, except that each instruction added by them is emitted. To illustrate how code is generated, consider the reductions that compiled code for an application MN will take when supplied with an environment e that supplies code generators for the free code variables in M and N, and which has a code block as its rightmost value.

$$\begin{array}{lll}
((e, \{P'\}) :: S, & \underline{\langle}[\![M]\!]_\Omega^\Lambda \underline{\mathtt{,}} [\![N]\!]_\Omega^\Lambda \underline{\rangle}; \underline{\mathtt{app}}; P) & \Longrightarrow \\
((e, \{P'; \langle\}) :: S, & [\![M]\!]_\Omega^\Lambda \underline{\mathtt{,}} [\![N]\!]_\Omega^\Lambda \underline{\rangle}; \underline{\mathtt{app}}; P) & \Longrightarrow^* \\
((e, \{P'; \langle P_M\}) :: S, & \underline{\mathtt{,}} [\![N]\!]_\Omega^\Lambda \underline{\rangle}; \underline{\mathtt{app}}; P) & \Longrightarrow \\
((e, \{P'; \langle P_M \mathtt{,}\}) :: S, & [\![N]\!]_\Omega^\Lambda \underline{\rangle}; \underline{\mathtt{app}}; P) & \Longrightarrow^* \\
((e, \{P'; \langle P_M \mathtt{,} P_N\}) :: S, & \underline{\rangle}; \underline{\mathtt{app}}; P) & \Longrightarrow \\
((e, \{P'; \langle P_M \mathtt{,} P_N \rangle\}) :: S, & \underline{\mathtt{app}}; P) & \Longrightarrow \\
((e, \{P'; \langle P_M \mathtt{,} P_N \rangle; \mathtt{app}\}) :: S, & P) &
\end{array}$$

Here, P_M is the program emitted by $[\![M]\!]_\Omega^\Lambda$ and P_N is the program emitted by $[\![N]\!]_\Omega^\Lambda$.

A technical problem arises when we compile functions in code generators. Since the argument of `emit` must be a simple instruction and therefore not a `Cur`, we cannot simply compile `code` $\lambda x.M$ as `emit(Cur(`$[\![M]\!]_\Omega^\Lambda$`))`. For this reason the CCAM has the special instruction `merge` which allows one segment of emitted code to be inserted into another as the argument of a `Cur`. This also means that we must be able handle multiple code blocks, and be sure that we will emit code to the appropriate one at the appropriate time. To compile an abstraction inside a code generator, we need to initialize a new code block, emit the body of the function to that code block, and the merge it back into the original code block.

$$[\![\lambda x.M]\!]_\Omega^\Lambda = \langle \langle \mathtt{fst,arena} \rangle; [\![M]\!]_\Omega^{(\Lambda,x)}; \mathtt{snd,id} \rangle; \mathtt{merge}$$

Some code variables in code generators are bound to code generators which should be called in order to emit their code into the current block, effectively performing code substitution. However, not all code variables encountered will be bound yet. For example, in the expression

```
let cogen u = code (4 + 5)
in code (let cogen u' = code 2
         in 3 + u' + u end) end
```

the term code (let cogen $u' =$ code 2 in $3+u'+u$ end) (call it M) contains to a reference to u which will have a code generator bound in the environment when M is activated as a code generator. However, u', which is also referenced in M will not have an entry in the environment until the code of M is generated and run. Therefore we need to take careful account of which variables are available and when. The compilation function for code generators, $\llbracket M \rrbracket_{\Omega}^{\Lambda}$ employs two variable contexts to do this: one (Λ) to hold the position of the variables which have bindings available at generation time, and to which we can therefore specialize the code; and the other (Ω) to hold the position of the rest of the variables. Thus, compiling a code variable depends on in which context the variable is bound.

$$\llbracket u \rrbracket_{\Omega}^{\Lambda} = \begin{cases} \underline{\langle \Omega(u)\texttt{,}\,\texttt{arena}\rangle}; \underline{\texttt{app}}; \underline{\texttt{call}} & \text{if } u \text{ is in } \Omega \\ \langle \texttt{fst}, \langle \texttt{fst}; \Lambda(u), \texttt{snd}\rangle; \texttt{app}; \texttt{snd}\rangle & \text{if } u \text{ is in } \Lambda \end{cases}$$

The first case, when u is in Ω, corresponds to compiling a code variable that does not yet have a binding in the environment (u' in the example above). It emits into the current block code which will activate the generating extension to which the variable will become bound. Therefore, it looks just like case for code variables in the normal compiler, except that each of its instructions is wrapped with an emit. The second case handles code variables which already have code generators available. These are activated and the code that they generate is added to the current code block.

Recall from Section 6.1 that we want to compile nested code generators into functions which we simply lift into the code block, without specialization. This function accepts as its argument an environment, and the lifted function is applied to the current environment. All of this makes for a complex rule for code M.

$$\begin{aligned} &\llbracket \texttt{code } M \rrbracket_{\Omega}^{\Lambda} = \\ &\underline{\langle}\langle \texttt{fst}, \langle \text{‘}(); \texttt{Cur}(\texttt{fst}; \texttt{Cur}(\llbracket M \rrbracket_{()}^{\Lambda|\Omega})), \texttt{snd}\rangle; \texttt{lift}; \texttt{snd}\rangle\underline{,} \\ &\quad \underline{\texttt{id}}\underline{\rangle}; \underline{\texttt{app}} \end{aligned}$$

To understand this complicated rule, first observe that the program $\langle P, \texttt{id}\rangle; \texttt{app}$ will be emitted into the current code block, where P is the program that is emitted by the code $\langle \texttt{fst}, \langle \text{‘}(); \texttt{Cur}(\texttt{fst}; \texttt{Cur}(\llbracket M \rrbracket_{()}^{\Lambda|\Omega})), \texttt{snd}\rangle; \texttt{lift}; \texttt{snd}\rangle$. This will have the effect of explicitly applying the function produced by P when it is run to the current environment, since id leaves the environment untouched.

P creates a closure containing the code for a generator for M and lifts that closure into the code block. In detail, P first saves a copy of the current environment without the current code block on the stack.

$$\begin{aligned} &((e, \{P'\}) :: S, \\ &\langle \texttt{fst}, \langle \text{‘}(); \texttt{Cur}(\texttt{fst}; \texttt{Cur}(\llbracket M \rrbracket_{()}^{\Lambda|\Omega})), \texttt{snd}\rangle; \texttt{lift}; \texttt{snd}\rangle; P) \\ &\quad \Longrightarrow^{*} \\ &((e, \{P'\}) :: e :: S, \\ &\langle \text{‘}(); \texttt{Cur}(\texttt{fst}; \texttt{Cur}(\llbracket M \rrbracket_{\Omega|\Lambda}^{()})), \texttt{snd}\rangle; \texttt{lift}; \texttt{snd}\rangle; P) \end{aligned}$$

Next, an empty environment is created, and the closure is created in that empty environment (the environment that the code generator will need will be supplied at the time the generator is activated).

$$\begin{aligned} &((e, \{P'\}) :: e :: S, \\ &\langle \text{‘}(); \texttt{Cur}(\texttt{fst}; \texttt{Cur}(\llbracket M \rrbracket_{()}^{\Lambda|\Omega}())), \texttt{snd}\rangle; \texttt{lift}; \texttt{snd}\rangle; P) \Longrightarrow^{*} \\[1ex] &(() :: (e, \{P'\}) :: e :: S, \\ &\texttt{Cur}(\texttt{fst}; \texttt{Cur}(\llbracket M \rrbracket_{()}^{\Lambda|\Omega})), \texttt{snd}\rangle; \texttt{lift}; \texttt{snd}\rangle; P) \Longrightarrow \\[1ex] &([() : \texttt{fst}; \texttt{Cur}(\llbracket M \rrbracket_{()}^{\Lambda|\Omega})] :: (e, \{P'\}) :: e :: S, \\ &, \texttt{snd}\rangle; \texttt{lift}; \texttt{snd}\rangle; P) \end{aligned}$$

Now the closure and the code block are paired and the closure is lifted into the block, and the resulting block is paired with the saved environment.

$$\begin{aligned} &([() : \texttt{fst}; \texttt{Cur}(\llbracket M \rrbracket_{()}^{\Lambda|\Omega})] :: (e, \{P'\}) :: e :: S, \\ &, \texttt{snd}\rangle; \texttt{lift}; \texttt{snd}\rangle; P) \Longrightarrow^{*} \\[1ex] &(([() : \texttt{fst}; \texttt{Cur}(\llbracket M \rrbracket_{()}^{\Lambda|\Omega})], \{P'\}) :: e :: S, \\ &\texttt{lift}; \texttt{snd}\rangle; P) \Longrightarrow^{*} \\[1ex] &(\{P'; \text{‘}[() : \texttt{fst}; \texttt{Cur}(\llbracket M \rrbracket_{()}^{\Lambda|\Omega})]\} :: e :: S, P) \end{aligned}$$

Thus, running $\llbracket M \rrbracket_{\Omega}^{\Lambda}$ will deposit

$$\langle \text{‘}[() : \texttt{fst}; \texttt{Cur}(\llbracket M \rrbracket_{()}^{\Lambda|\Omega})], \texttt{id}\rangle; \texttt{app}$$

in the current code block, which will, when run in an environment e create the closure $[e : \texttt{Cur}(\llbracket M \rrbracket_{()}^{\Lambda|\Omega})]$, which is a code generator.

The compilation functions are summarized in Figure 4. Many optimizations are possible, but left out of the figure in the interests of brevity.

7 Implementation

We have implemented a prototype $\text{ML}^{\Box}$ compiler. The language includes most of core ML, including datatypes, reference cells, and arrays. All of the programs presented in this paper are working programs compiled by our compiler. The compiler generates code for the CCAM extended with support for efficient handling of conditionals, recursion, and various base types.

In addition, we have built a CCAM simulator on which to run the output of our compiler. While CCAM instructions are rather abstract compared to native machine code, we can still observe the benefits of specialization by counting reduction steps in the CCAM machine. Figure 1 contains a list of reduction counts for executions of the CCAM on some of the programs in this paper.

As indicated in the table, the non-specializing packet filter interpreter, evalpf, takes 9163 reductions on our CCAM simulator to recognize each telnet packet presented to it. On the other hand the specializing interpreter, bevalpf, only takes 1104 reduction steps after paying an initial cost of 11984 reductions to generate a specialized version of the interpreter for the telnet packet filter.

We also obtain a significant speedup from the specializing version of the polynomial calculator, compPoly. In fact, the specialized version somewhat suspiciously pays for itself after only one application. However, part of the improvement in performance of compPoly over evalPoly comes from the fact that compPoly is impeded less by the low quality of our current pattern matching compiler.

$$
\begin{array}{lcl}
\llbracket x \rrbracket_\Omega & = & \Omega(x) \\
\llbracket \lambda x.M \rrbracket_\Omega & = & \mathtt{Cur}(\llbracket M \rrbracket_{(\Omega,x)}) \\
\llbracket M N \rrbracket_\Omega & = & \langle \llbracket M \rrbracket_\Omega \mathtt{,} \llbracket N \rrbracket_\Omega \rangle ; \mathtt{app} \\
& & \\
\llbracket u \rrbracket_\Omega & = & \langle \Omega(u)\mathtt{,arena} \rangle ; \mathtt{app}; \mathtt{call} \\
\llbracket \mathtt{code}\ M \rrbracket_\Omega & = & \mathtt{Cur}(\llbracket M \rrbracket_\Omega^{()}) \\
\llbracket \mathtt{lift}\ M \rrbracket_\Omega & = & \llbracket M \rrbracket_\Omega ; \mathtt{Cur(lift)} \\
\llbracket \mathtt{let\ cogen}\ u = M\ \mathtt{in}\ N\ \mathtt{end} \rrbracket_\Omega & = & \langle \llbracket M \rrbracket_\Omega \rangle ; \llbracket N \rrbracket_{(\Omega,u)} \\
& & \\
\llbracket x \rrbracket_\Omega^\Lambda & = & \underline{\Omega(x)} \\
\llbracket \lambda x.M \rrbracket_\Omega^\Lambda & = & \langle \langle \mathtt{fst,arena} \rangle ; \llbracket M \rrbracket_{(\Omega,x)}^\Lambda ; \mathtt{snd,id} \rangle ; \mathtt{merge} \\
\llbracket M N \rrbracket_\Omega^\Lambda & = & \underline{\langle} \llbracket M \rrbracket_\Omega^\Lambda \underline{\mathtt{,}} \llbracket N \rrbracket_\Omega^\Lambda \underline{\rangle} ; \underline{\mathtt{app}} \\
& & \\
\llbracket u \rrbracket_\Omega^\Lambda & = & \left\{ \begin{array}{ll} \underline{\langle \Omega(u)}\underline{\mathtt{,arena}\rangle} ; \underline{\mathtt{app}} ; \underline{\mathtt{call}} & \text{if } u \text{ is in } \Omega \\ \langle \mathtt{fst,} \langle \mathtt{fst}; \Lambda(u)\mathtt{,snd} \rangle ; \mathtt{app}; \mathtt{snd} \rangle & \text{if } u \text{ is in } \Lambda \end{array} \right. \\
\llbracket \mathtt{code}\ M \rrbracket_\Omega^\Lambda & = & \underline{\langle} \langle \mathtt{fst,} \langle \text{`}(); \mathtt{Cur(fst;Cur(}\llbracket M \rrbracket_{()}^{\Lambda|\Omega}\mathtt{)),snd} \rangle ; \mathtt{lift}; \mathtt{snd} \rangle \underline{\mathtt{,id}\rangle} ; \underline{\mathtt{app}} \\
\llbracket \mathtt{lift}\ M \rrbracket_\Omega^\Lambda & = & \llbracket M \rrbracket_\Omega^\Lambda ; \langle \langle \mathtt{fst,arena} \rangle ; \underline{\mathtt{lift}} ; \mathtt{snd,id} \rangle ; \mathtt{merge} \\
\llbracket \mathtt{let\ cogen}\ u = M\ \mathtt{in}\ N\ \mathtt{end} \rrbracket_\Omega^\Lambda & = & \underline{\langle} \llbracket M \rrbracket_\Omega^\Lambda \underline{\rangle} ; \llbracket N \rrbracket_{(\Omega,u)}^\Lambda
\end{array}
$$

Figure 4: Compilation rules

Computation	Reductions
`evalpf` on first telnet packet	9163
`evalpf` on n^{th} telnet packet	9163
`bevalpf` on first telnet packet	11984
`bevalpf` on n^{th} telnet packet	1104
`evalPoly` (47, `poly1`)	807
`specPoly poly1`	443
`poly1Target` 47	175
`compPoly poly1`	553
`eval codeGenerator`	200
`mlPolyFun` 47	74

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

8 Related Work

The style of run-time specialization described in Section 5.1 is inspired by the Fabius system (Lee and Leone 1996). Fabius compiles a pure, first-order subset of ML into native MIPS code. It chooses curried functions as the sites for specialization, compiling them into generating extensions parameterized by their early arguments.

Tempo (Consel and Noël 1996) is a compiler which extends several techniques from traditional partial evaluation in order to stage C programs semi-automatically. Programmers can supply an initial division of a program into stages via auxiliary data files which are used to guide a binding-time analysis. Programs are then compiled into templates which can be specialized either at run time or in a pre-run time specialization phase. The DyC system (Auslander, Philipose, Chambers, Eggers, and Bershad 1996; Grant, Mock, Philipose, Chambers, and Eggers 1997) provides a relatively sophisticated and declarative annotation scheme for specifying the initial division of inputs, but then shares several characteristics with Tempo, including its use of an extended binding-time analysis and use of template filling for run-time specialization.

Though we have chosen to compile $\mathrm{ML}^\Box$ programs to perform run-time code generation in the style of Fabius, it is easy to see that we could also view `code` expressions as templates, and their free code variables as the holes to be filled. Instruction-stream encoding and template filling each have their advantages as techniques for run-time code generation. It is conceivable that a sophisticated compiler might have both techniques at its disposal and use the most appropriate one based upon the requirements of the program.

Engler, Hsieh, and Kaashoek (1996) describe 'C, an extension of the C language which has facilities for creating run-time specializing programs. 'C has an annotation scheme in the style of Lisp which allows programmers to annotate their programs to stage them the way they desire. The compiler for ‘C, called tcc, provides several different back ends, depending on the style of run-time specialization desired. These styles range from a heavyweight invocation of a full C compiler back-end at run time, to a lighter-weight single-pass translation to an abstract machine code.

Davies (1996) describes the language $\lambda^\bigcirc$, which is based on linear-time temporal logic. It allows the manipulation of code with free variables and can thereby force inlining, which is not expressible in $\mathrm{ML}^\Box$. However, an `eval` operator is not expressible in this language—partial evaluation can proceed only by a sequence of global program transformations.

Taha and Sheard (1997) attempt to combine the benefits of $\mathrm{ML}^\Box$ and $\lambda^\bigcirc$ in the language MetaML. The language can express inlining in much the same way as $\lambda^\bigcirc$, while still allowing an `eval` operator. However, MetaML is not known to have a logical basis as the other two languages. Furthermore, their operational semantics is quite different from the CCAM and does not address light-weight run-time code generation.

9 Conclusion

We have designed the language $\mathrm{ML}^\Box$, and implemented a compiler that translates compiles $\mathrm{ML}^\Box$ `code` expressions into efficient run-time code generators. The compiler targets the CCAM, an extension of the CAM carefully designed to emulate the style of run-time code generation first provided by the Fabius compiler.

In our early experience with the ML$^\Box$ language and our compiler, we have been able to express precisely the staging of computations necessary to take best advantage of the run-time code generation facilities of the CCAM. This experience is an early indication that a language that provides explicit control over staging decisions can be a practical way to improve the performance of programs. Furthermore, we have found that the typed framework of Mini-ML$_e^\Box$ provides, in addition to an expressive method for specifying staged computation, a basis for formalization of both ML$^\Box$ and its compiler. We believe that this framework extends to other languages and compilation schemes, and thus provides a basis for reasoning about staged computation in general.

Acknowledgments

We would like to thank Rowan Davies, Olivier Danvy, and the anonymous reviewers for their comments.

References

Auslander, J., M. Philipose, C. Chambers, S. Eggers, and B. Bershad (1996, May). Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, Pennsylvania.

Consel, C. and F. Noël (1996, 21–24 January). 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*, pp. 145–156.

Cousineau, G., P.-L. Curien, and M. Mauny (1987). The categorical abstract machine. *Science of Computer Programming*, 173–202.

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

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

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

Engler, D. R., D. Wallach, and M. F. Kaashoek (1995, March).
Efficient, safe, application-specific message processing. Technical Memorandum MIT/LCS/TM533, MIT Laboratory for Computer Science.

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

Grant, B., M. Mock, M. Philipose, C. Chambers, and S. J. Eggers (1997). DyC: An expressive annotation-directed dynamic compiler for C. Technical Report UW-CSE-97-03-03, Department of Computer Science, University of Washington.

Jørring, U. and W. L. Scherlis (1986, 21–24 January). Compilers and staging transformations. In *Conference Record of POPL '86: The thth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 86–96.

Jones, N. D., C. K. Gomard, and P. Sestoft (1993). *Partial Evaluation and Automatic Program Generation*. Prentice-Hall.

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

Kohlbecker, E., D. P. Friedman, M. Felleisen, and B. Duba (1986). Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pp. 151–159.

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

Leone, M. and P. Lee (1998). Dynamic specialization in the Fabius system. *ACM Computing Surveys 1998 Symposium on Partial Evaluation*. To appear.

Massalin, H. and C. Pu (1989, December). Threads and input/output in the Synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pp. 191–201.

McCanne, S. and V. Jacobson (1993, January). The BSD packet filter: A new architecture for user-level packet capture. In *The Winter 1993 USENIX Conference*, pp. 259–269. USENIX Association.

Pike, R., B. Locanthi, and J. Reiser (1985, February). Hardware/software trade-offs for bitmap graphics on the Blit. *Software — Practice and Experience 15*(2), 131–151.

Sirer, E. G., S. Savage, P. Pardyak, G. P. DeFouw, and B. N. Bershad (1996, February). Writing an operating system with Modula-3. In *The Inaugural Workshop on Compiler Support for Systems Software*, pp. 134–140.

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

Wickline, P., P. Lee, F. Pfenning, and R. Davies (1998). Modal types as staging specifications for run-time code generation. *ACM Computing Surveys 1998 Symposium on Partial Evaluation*. To appear.