Run-time Code Generation and Modal-ML

Philip Wickline Peter Lee Frank Pfenning

January, 1998

CMU-CS-98-100

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also appears as CMU-CS-FOX-98-01. A revised version of this paper will appear in the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation.

This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050, and in part by the National Science Foundation under grant#CCR-9619832 .

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the National Science Foundation, the Advanced Research Projects Agency, or the U.S. Government

Abstract

This paper presents early experience with a typed programming language and compiler for run-time code generation. The language is an extension of the SML language with modal operators, based on the $\lambda^{\Box}$ language of Davies and Pfenning. It allows programmers to specify precisely, through types, the stages of computation in a program. The compiler generates target code that makes use of run-time code generation in order to exploit the staging information. The target machine is currently a version of the Categorial Abstract Machine, called the CCAM, which we have extended with facilities for run-time code generation. Using this approach, the programmer is able to express the staging that he wants to the compiler directly. 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 to generate 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 $\lambda^{\Box}$ 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.

Keywords: Modal logic, Run-time code generation, Generating extensions, Partial evaluation

1 Introduction

In this paper, we present a programming language that allows programmers to specify stages of computation in a program, along with an implementation technique based on run-time code generation for exploiting the staging.

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 in later computations in a way that leads to faster execution. To achieve this effect, programmers often stage their program manually, using *ad hoc* methods; there have also been some attempts to make such staging transformations more systematic [16]. Another approach, used in partial evaluation [8], is to automate the staging of programs 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. More recent work has extended the partial evaluation framework to account for multiple computation stages [7].

In recent years, several researchers have studied the use of run-time code generation (RTCG) to exploit staged computation [1, 3, 9, 10, 12]. One advantage of RTCG is that opens the possibility of low-level code optimizations (such as register allocation, instruction selection, loop unrolling, array-bounds checking removal, and so on) to take advantage of values that are not known until run time. Such optimization cannot normally be expressed by a source-to-source transformation

In order to make use of RTGC, 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 (notably the Tempo system [1]), we are interested here in developing a programming language that supports a systematic method for describing the computation stages. Besides providing the programmer with full control over when and where RTCG occurs, we believe the overall implementation should also become much simpler since the complexity of a sophisticated automatic analysis can be avoided.

The idea of using a programming notation for staging is far from new. The backquote and antiquote notation of Lisp macros, for example, provides an intuitive though highly error-prone approach to staged computation. More recent annotation schemes used by RTCG systems include that of 'C [3] and Fabius [10]. These languages allow the programmer to communicate his intentions to the compiler in a relatively straightforward manner. Unfortunately, in the case of the Fabius system, the annotation scheme is extremely simple, thus limiting the ability of the programmer to express staging decisions. The difficulty in 'C (and Lisp), on the other hand, is that there is no direct way to ensure that the staging behavior which the programmer specifies is correct: programs can be (and are) written that will result in run-time errors. Such errors include referencing a variable that is not yet available, and referencing variables which are no longer available.

We propose that an extension of the SML language and type system can be used as a clear and expressive notation for staged computation. Drawing on previous work on the language $\lambda^{\Box}$ [2] which is based on the modal logic S4, and on the interpretation of this language for run-time code generation described in [18], we present an implementation of a prototype compiler for a version of the SML language (without modules) that uses modal operators to specify early and late stages of a program's computation. We then apply compilation techniques patterned after those developed for the Fabius system [10] in order to compile programs into code that performs RTCG according to the mode of each subexpression in the program. 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 indirectly through a heavyweight (and usually unpredictable) analysis.
- The programmer is given a framework which al-

lows 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 implementation. 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 any other means at its disposal.

- The language naturally handles situations in which more than two stages are desired, such as Fabius-style multi-stage specialization [11]. This arises, for example, when dynamically generated code can used to compute values that are used in the dynamic specialization of yet more code.

In order to demonstrate these advantages, we have implemented a prototype compiler for the Standard ML language (without modules), extended with modal operators and types. The compiler generates code for a version of the Categorial Abstract Machine [6], called the CCAM, which is extended with a facility for emitting fresh code at run time.

We begin the paper with a brief introduction to the $\lambda^{\Box}$ language, on which our dialect of SML is based. Then, we give a series of program 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 Fabius-style multi-stage specialization. Next, we present the CCAM, followed by a description of how $\lambda^{\Box}$ programs are compiled into CCAM code. Finally, we discuss some of the details of the actual implementation of our compiler, and present some benchmark results to show how staged programs can lead to better performance.

Types	A, B	$::=$	$A \to B \mid \Box A$
Terms	M, N	$::=$	$x \mid \lambda x.M \mid M N$ $\mid u \mid \texttt{code}\ M \mid \texttt{lift}\ M$ $\mid \texttt{let cogen}\ u = M\ \texttt{in}\ N$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, x : A \mid \Gamma, u : A$

Figure 1: $\lambda^{\Box}$ Syntax

2 The Modal Lambda-Calculus

We briefly introduce the language $\lambda^{\Box}$ which is a simplification of the explicit version of $\mathrm{ML}^{\Box}$ described in Davies and Pfenning [2]. Although we present only $\lambda^{\Box}$ here because of space considerations, the compilation technique described in the section 5 extends easily to all core SML constructs. Indeed, we have implemented a prototype compiler for most of core ML extended with the modal constructs.

2.1 Syntax

$\lambda^{\Box}$ arises from the simply-typed λ-calculus by adding a new type constructor $\Box$. Except for the addition of *lift*, 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*".

In our context, we think of $\Box A$ as the type of *generators for code of type A*. Generators are created with the `code` M construct. For example, $\vdash \texttt{code}\ (\lambda x.x) : \Box(A \to A)$ is a generator which, when invoked, generates code for the identity function and then calls it. Figure 1 presents the syntax of $\lambda^{\Box}$. 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). Their role is explained below.

To invoke a generator, one might expect a corresponding *eval* construct of type $(\Box A) \to A$. Such a function is in fact definable, but not a suitable basis for the language. Instead we have a binding construct $\texttt{let cogen}\ u = M\ \texttt{in}\ N$ which expects a code generator M of type $\Box A$ and binds a *code variable* u (which we will sometimes call a *modal variable*). However, even evaluation of `let cogen` $u = M$ `in` N *will not immediately generate code.*

Code will not be emitted until the modal variable u is encountered during evaluation. For example,

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

is the function *eval* mentioned above which invokes a generator and to create new code, 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.

$$\begin{array}{l}\vdash \lambda f.\lambda g.\texttt{let cogen } f' = f \texttt{ in} \\ \quad \texttt{let cogen } g' = g \texttt{ in code } \lambda x.f'(g'(x)) \\ \qquad : \Box(B \rightarrow C) \rightarrow \Box(A \rightarrow B) \rightarrow \Box(A \rightarrow C)\end{array}$$

Readers familiar with $\text{ML}^{\Box}$ will notice that we have added the operator *lift*, which obeys the rule that `lift` M has type $\Box A$ if M has type A. `lift` M evaluates M and returns a generator which just "quotes" the resulting value. In contrast to *code* this prohibits all optimizations during code generation. As noted in Davies and Pfenning [2], *lift* is definable in $\text{ML}^{\Box}$ for base types, but its general form has no logical foundation. Here we show that it nonetheless has a reasonable and useful operational interpretation in the context of run-time code generation.

2.2 Typing Rules

The typing rules, presented in figure 2, 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*", "*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. The *let boxed* rule expresses that if we have a value which is a code generator (and therefore of type $\Box A$), we can bind a code variable u of type A which may be included in other code generators.

3 Programming with $\text{ML}^{\Box}$

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

3.1 Computing the Value of Polynomials

To start with a simple example, consider the following ML function which evaluates a given polynomial for a given base. For this function, the polynomial $a_0 + a_1 x + a_2 x^2 + \cdots a_n x^n$ 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. 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
```

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

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

$$\frac{u : A \text{ in } \Delta}{\Delta; \Gamma \vdash u : A} \qquad \frac{\Delta; \Gamma \vdash M : A}{\Delta; \Gamma \vdash \texttt{lift}\ M : \Box A}$$

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

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

```
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. Therefore we rewrite `specPoly` as the $\mathrm{ML}^{\Box}$ 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 the postfix type constructor $ is the $\Box$ type. 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.

3.2 Libraries

Suppose we were to build a library of useful functions. One possibility afforded by $\mathrm{ML}^{\Box}$ is to install staged versions of the 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 application program:

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

Even though the `client` program does not have access to the source code of `compPoly` library routine, it is still able to benefit from the fact that it will perform RTCG on the polynomial computed by `makePoly` (which presumably has type `t2 -> poly`).

This example also illustrates one way multi-stage specialization can be achieved in our system. Note that the `client` program takes the argument `x` and generates code for a `t2 -> t3` function, 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 extremely difficult to achieve in standard partial evaluation, but falls out naturally in our framework.

3.3 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. As a result, many useless packets may be delivered, with a consequent degradation of performance. 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 [15, 14]. 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 into safe native code [5, 10, 13, 17]. To demonstrate this idea in our language, consider the following excerpt of the implementation of a simple interpreter for the BSD packet filter language [14] in SML.

```
(* 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.

```
(* 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 6, we show that the improvement in execution time for a typical BPF packet filter is substantial.

3.4 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 time, making the result program *slower* rather than faster. We must carefully arrange to have generated programs saved for future use in situations where we think are likely to be needed again. Fortunately, we can bind up this functionality with the function itself.

```
(*
specCode : (int, int -> int) table
get : ('a, 'b) table * 'a -> 'b option
add : ('a, 'b) table * ('a * 'b) -> 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 hash 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 won't have to do any additional work to make a generating extension for the second call.

```
(*
specCode : (int, int -> int) table
genExts : (int, (int -> int)$) table
get : ('a, 'b) table * 'a -> 'b option
add : ('a, 'b) table * ('a * 'b) -> 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 (genExt, 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 excessively tedious in some cases, it does allow the programmer to very carefully control what and how memoization will occur. Furthermore, generic memoization routines could be written that can easily accomodate most common memoization needs.

4 The CCAM

' In this section we present the CCAM, an *ad-hoc* extension of the CAM[6] which provides facilities for run-time code generation and which we use as the target of the compiler detailed in the next section.

4.1 Fabius and Run-Time Code Generation

The Fabius compiler[10] delivers dramatic speedups over conventional compilers for some programs by compiling selected functions in its input to generating extensions. Using values obtained in earlier computations, these generating extensions create code specialized to perform later computations. While several different schemes for run-time code generation have been used in other systems [4, 3, 13, 12, 1] Fabius is able to achieve a remarkably low instruction-executed-to-instruction-specialized ratio by a unique combination of features.

- Generating extensions produced by Fabius never manipulate source-level terms at run time. Instead machine language programs are synthesized directly from machine language programs. Fabius in not unique in this respect: the Synthesis kernel [13, 12] and Tempo compiler[1] also share this property.
- Fabius encodes terms to be specialized directly into the instruction stream, usually in the form of immediate operands to instructions. This is in contrast to systems which copy templates and fill in holes at run time, such as Tempo and the Synthesis kernel. Instruction stream encoding allows Fabius to be very flexible about the kinds of specialization it can arrange to have performed at run time.
- Programs compiled by Fabius allows dynamic staging of code, i.e. the number of times that a program specializes itself may be dependent 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. Conventional off-line partial evaluation will fail to serve such a program well because even multi-level partial eval-

uation has no way to specialize on each of the variably many inputs.

4.2 An Abstract Machine for Run-time Code Generation

While developing the compilation technique for $\mathrm{ML}^{\Box}$ we wanted to compile programs to include generating extensions that have the same three properties that we list above for Fabius. We also thought it desirable to abstract away as much as possible from the details of individual architectures. However, since we want to create generating extensions that do not manipulate source level terms, but instead generates machine instructions directly, details of the machine to which we compile must find their way into our translation scheme. For this reason we developed the CCAM. We believe it to be a reasonable formalism that provides that capabilities that we need, while hiding details about individual architectures and instruction sets.

The primary novelty of the CCAM is the $\mathtt{emit}(i)$ instruction. It is intended to represent 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. This is designed to emulate the technique of run-time instruction encoding used in the Fabius compiler.

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 to a series of instructions which would test 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.

That we wish to produce multi-staged programs is a potential problem for our abstract machine. If we encode generating extensions with $\mathtt{emit}(i)$ instructions, must a program which contains a generating extension which produces code which is itself a generating extension give rise to instructions of the form $\mathtt{emit(emit(}i\mathtt{))}$? 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 $\overbrace{\mathtt{emit(emit(}\cdots\mathtt{emit}}^{n}(i)\cdots))$. For this reason, nested emits are not allowed on the CCAM, and our compilation scheme needs to take special steps in order to allow multi-level specialization. We show how to do this in section 5.

4.3 Instructions

The CCAM has the usual seven instructions associated with the CAM, and five more for code generation. $\mathtt{emit}(i)$, which has already been described, creates the instruction i in a new, dynamically created code sequence, called an arena. The `lift` instruction residualizes a value into an arena. `arena` creates a new arena, while `call` inserts dynamically generated code from an arena into the current instruction stream. Finally `merge` merges two arenas by inserting one as a function in the other.

Simple Inst	i	::=	$\mathtt{id} \mid \mathtt{fst} \mid \mathtt{snd} \mid \mathtt{push}$ $\mid \mathtt{swap} \mid \mathtt{cons} \mid \mathtt{app}$ $\mid `v \mid \mathtt{lift} \mid \mathtt{arena}$ $\mid \mathtt{merge} \mid \mathtt{call}$
Composite Inst	I	::=	$i \mid \mathtt{emit}(i) \mid \mathtt{Cur}(P)$
Values	v, u	::=	$(v, u) \mid [v : P] \mid B \mid ()$
Code Blocks	B	::=	$\{P\}$
Sequences	P	::=	$\cdot \mid I; P$
Stacks	S	::=	$\cdot \mid v :: S$

4.4 Transitions

A configuration, $\langle S, P\rangle$, of the CCAM consists of a stack of values and an instruction sequence, representing the current instruction stream. We will routinely omit the final $\cdot$ on stacks and instruction sequences. We use $P'@P$ to represent the obvious sequence obtain by appending the sequences P to the sequence P'. Figure 3 lists the transitions of the CCAM.

5 Compilation

The translation from $\lambda^{\Box}$ to CCAM code is detailed in this section. The translation is divided into two parts: translation of code which is not initially inside a code generator, and the translation of code generators. These two translations are represented by $[\![M]\!]_E$ and $[\![M]\!]^E_{LE}$.

$[\![M]\!]_E$ denotes the translation of non-code-generating code M in a context E, which simply describes the location of variables in the run-time environment. Variable contexts are built from variables and the empty context as follows:

$$\text{VariableEnvironments} \quad E, LE \quad ::= \quad \bullet \mid E, u$$

To save space and for convenience we will often write `emit`(i) as $\underline{\text{i}}$, and the pairing operators `push`,`swap`, and `cons` as '$\langle$', ',', and '$\rangle$', respectively.

The rules for translating applications, non-modal variables, and abstractions in a non-code-generating context are the same as those in [6]. We compile `code` expressions to generating extensions, which are functions from arenas to arenas. An extension emits its code into its argument arena, and returns that transformed arena. Modal variables must select out of the environment the generating extension to which it is bound, and apply the extension to a new arena, and then finally jump to the newly created code. Thus, it is when modal variables are referenced outside of `code` constructs that code generation actually occurs. Finally, the `let cogen` construct translates to code which augments the environment with the result of the bound expression and then executes the body of the expression.

The $[\![M]\!]^E_{LE}$ relation compiles a $\lambda^{\Box}$ term into a generating extension. It uses two contexts, an "early" context E which will hold the location of variables in the environment from all stages, and a "late" context LE, which is really just a pointer into the early context that marks the division between variables available at generation time and those which will only be available later when the generated code is run. The translation rules for applications and non-modal variables underneath `code` constructors are similar to those for their non-code-generating relatives, except the instructions are buried under `emit()` instructions. The abstraction rule, on the other hand, is complicated considerably by the fact that the argument of a `Cur` is a sequence of instructions, and instructions must be emitted individually. This is the reason for the `merge` instruction. It enables us to emit code to a new arena and then treat that code as the body of a function. Implemented on a real computer, this would correspond to the fact that the text of a function is typically stored in a separate area, and a function call involves jumping to the location of the function.

Translating modal variables under `code` constructors depends on where the variable is bound. If it is bound under the same `code` constructor in which the variable finds itself, then there is no generating extension yet available in the environment for it, and so it must be rebuilt as a reference to its binder. If, on the other hand, it is bound outside the `code` constructor, then it should be applied to the current arena, thereby effectively substituting its code into the current code.

The primary difficulty in the compilation is avoiding nested emits. We achieve this by arranging to have generating extensions specialize all of the code that they contain, *except* the code for other generating extensions. This results in a rather complicated looking case in the compilation for `code` expressions under other `code` constructors. Essentially, the code arranges to have a closure containing the body of the `code` expression inserted into the arena. This closure is explicitly applied to the "late" environment so that it can access all the variables bound within it.

The boxed `lift` and `let cogen` rules are mostly emitted versions of the unboxed forms, except that

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

Figure 3: Transitions of the CCAM

the `lift` rules needs to go through the same contortions as abstractions to insert a `Cur` into the arena.

6 ML$^{\Box}$ compiler

We have implemented a prototype ML$^{\Box}$ compiler, for a large subset of core ML, including datatypes, refence cells, and arrays, extended with the modal constructs. All of the programs presented in this paper are working programs compilable by our compiler. The compiler generates code for the CCAM extended with support for conditionals, recursion, and various base types.

In addition, we have built a CCAM simualtor 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 CCAM programs.

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

7 Conclusion

We have designed and implemented a compiler for the language ML$^{\Box}$ which compiles `code` expressions into

$$
\begin{array}{lcl}
get(a,(E,a)) & = & \mathtt{snd} \\
get(a,(E,b)) & = & \mathtt{fst}; get(a,E) \\
\\
\underline{get}(a,(E,a)) & = & \underline{\mathtt{snd}} \\
\underline{get}(a,(E,b)) & = & \underline{\mathtt{fst}}; \underline{get}(a,E) \\
\\
\llbracket x \rrbracket_E & = & get(x,E) \\
\llbracket \lambda x.M \rrbracket_E & = & \mathtt{Cur}(\llbracket M \rrbracket_E) \\
\llbracket MN \rrbracket_E & = & \langle \llbracket M \rrbracket_E, \llbracket N \rrbracket_E \rangle; \mathtt{app} \\
\\
\llbracket u \rrbracket_E & = & \langle get(u,E), \mathtt{arena} \rangle; \mathtt{app}; \mathtt{call} \\
\llbracket \mathtt{code}\ M \rrbracket_E & = & \mathtt{Cur}(\llbracket M \rrbracket^E_\bullet) \\
\llbracket \mathtt{lift}\ M \rrbracket_E & = & \llbracket M \rrbracket_E; \mathtt{Cur}(\mathtt{lift}) \\
\llbracket \mathtt{let\ cogen}\ u = M\ \mathtt{in}\ N \rrbracket_E & = & \langle \llbracket M \rrbracket_E \rangle; \llbracket N \rrbracket_{(E,u)} \\
\\
\llbracket x \rrbracket^E_{LE} & = & \underline{get}(x,LE) \\
\llbracket \lambda x.M \rrbracket^E_{LE} & = & \langle \langle \mathtt{fst}, \mathtt{arena} \rangle; \llbracket M \rrbracket^{(E,x)}_{(LE,x)}; \mathtt{snd}, \mathtt{id} \rangle; \mathtt{merge} \\
\llbracket MN \rrbracket^E_{LE} & = & \underline{\langle} \llbracket M \rrbracket^E_{LE} \underline{,} \llbracket N \rrbracket^E_{LE} \underline{\rangle}; \underline{\mathtt{app}} \\
\\
\llbracket u \rrbracket^E_{LE} & = & \left\{ \begin{array}{ll} \underline{\langle} \underline{\mathtt{fst}}; \underline{get}(u,LE) \underline{,} \underline{\mathtt{arena}} \underline{\rangle}; \underline{\mathtt{app}}; \underline{\mathtt{call}} & \text{if } u \text{ is in } LE \\ \langle \mathtt{fst}, \langle \mathtt{fst}; get(u,E), \mathtt{snd} \rangle; \mathtt{app}; \mathtt{snd} \rangle & \text{otherwise} \end{array} \right. \\
\llbracket \mathtt{code}\ M \rrbracket^E_{LE} & = & \underline{\langle} \langle \mathtt{fst}, \langle `(); \mathtt{Cur}(\mathtt{snd}; \mathtt{Cur}(\llbracket M \rrbracket^E_\bullet)), \mathtt{snd} \rangle; \mathtt{lift}; \mathtt{snd} \rangle \underline{,} \underline{\mathtt{id}} \underline{\rangle}; \underline{\mathtt{app}} \\
\llbracket \mathtt{lift}\ M \rrbracket^E_{LE} & = & \overline{\llbracket} M \rrbracket^E_{LE}; \langle \langle \mathtt{fst}, \mathtt{arena} \rangle; \underline{\mathtt{lift}}; \mathtt{snd}, \mathtt{id} \rangle; \mathtt{merge} \\
\llbracket \mathtt{let\ cogen}\ u = M\ \mathtt{in}\ N \rrbracket^E_{LE} & = & \underline{\langle} \llbracket M \rrbracket^E_{LE} \underline{\rangle}; \llbracket N \rrbracket^{(E,u)}_{(LE,u)}
\end{array}
$$

Figure 4: Compilation rules

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.

References

[1] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, 21–24 January 1996.

[2] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 258–270, 21–24 January 1996.

[3] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 131–144, 21–24 January 1996.

[4] Dawson R. Engler and Todd A. Proebsting. 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)*, pages 263–272. ACM Press, October 1994.

[5] Dawson R. Engler, Deborah Wallach, and M. Frans Kaashoek. Efficient, safe, application-specific message processing. Technical Memorandum MIT/LCS/TM533, MIT Laboratory for Computer Science, March 1995.

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

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

[8] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[9] David Keppel, Susan J. Eggers, and Robert R. Henry. Evaluating runtime-compiled value-specific optimizations. Technical Report 93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.

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

[11] Mark Leone and Peter Lee. Dynamic specialization in the fabius system. *ACM Computing Surveys 1998 Symposium on Partial Evaluation*, 1998.

[12] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Department of Computer Science, Columbia University, 1992.

[13] Henry Massalin and Calton Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[14] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *The Winter 1993 USENIX Conference*, pages 259–269. USENIX Association, January 1993.

[15] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *ACM Symposium on Operating Systems Principles*, pages 39–51. ACM Press, November 1987. An updated version is available as DEC WRL Research Report 87/2.

[16] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Conference Record of POPL '86: The th$^{\mathrm{th}}$ ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 86–96, 21–24 January 1986.

[17] Emin Gun Sirer, Stefan Savage, Przemyslaw Pardyak, Greg P. DeFouw, and Brian N. Bershad. Writing an operating system with Modula-3. In *The Inaugural Workshop on Compiler Support for Systems Software*, pages 134–140, February 1996.

[18] Philip Wickline, Peter Lee, Frank Pfenning, and Rowan Davies. Modal types as staging specifications for run-time code generation. *ACM Computing Surveys 1998 Symposium on Partial Evaluation*, 1998.