Dynamically Adaptable Software with Metacomputations in a Staged Language

Bill Harrison and Tim Sheard

Oregon Graduate Institute of Science and Technology
Beaverton, Oregon 97006, USA
`wlh@cse.ogi.edu,sheard@cse.ogi.edu`,
WWW home page: `http://www.cse.ogi.edu/~{wlh,sheard}`

Abstract

Profile-driven compiler optimizations take advantage of information gathered at runtime to re-compile programs into more efficient code. Such optimizations appear to be more easily incorporated within a semantics-directed compiler structure than within traditional compiler structure. We present a case study in which a metacomputation-based reference compiler for a small imperative language converts easily into a compiler which performs a particular profile-driven optimization: *local register allocation*. Our reference compiler is implemented in the staged, functional language MetaML and takes full advantage of the synergy between metacomputation-style language definitions and the staging constructs of MetaML. We believe that the approach to implementing profile-driven optimizations presented here suggests a useful, formal model for dynamically adaptable software.

1 Introduction

Dynamically adaptable software—software which can reconfigure itself at runtime in response to changes in the environment—is the focus of much current interest [2, 13, 18]. A classic example is the Synthesis Kernel [19] which dynamically specializes operating system kernel code to take advantage of runtime information. Staged programming [24] provides high-level abstractions for modeling such behavior. In this paper, we consider a particular kind of dynamically adaptable software, namely programming language compilers with profile-driven dynamic recompilation. The idea is to use staged programming to build a compiler for a language where the compiled code may periodically re-compile itself to take advantage of runtime information. A compiler constructed using staged programming does not have to rely on *ad hoc* techniques for specifying such optimizations.

Profile-driven compiler optimizations use information about a program's runtime behavior gathered during test executions to re-compile the program into more efficient code. An example of a profile-driven optimization is *local register allocation* [15, 1]. The idea behind this optimization is that, if the memory location bound to a program variable `x` is accessed sufficiently often during testing runs of the program, then it may improve the runtime performance of the program to recompile it with `x` stored in a register because access time for `x` would be reduced. This "usage count" heuristic may also be used to guide the inlining of procedures.

This paper makes two contributions. (1) We present a case study which suggests that profile-driven optimizations can be remarkably straightforward to implement in a semantics-directed compilation scheme based on metacomputations [7, 6]. We will demonstrate that the metacomputation structure makes it possible to achieve interoperability between static computation (e.g., program compilation) and dynamic computation (e.g., program execution), and we exploit this ability to implement a compiler which performs local register allocation. (2) We believe that the use of a staged language to construct code that reconfigures itself in response to dynamic stimuli illustrates an instance of a general model for constructing dynamically adaptable software in a concise and high level manner.

1.1 Compilation as Staged Computation

The usual semantics-directed paradigm is to write an interpreter and stage it, thereby producing a compiler. Typically, one may use a partial evaluator [11, 3] or a staged language [23, 21] to accomplish this. A compiler based on *metacomputations* and staging [7, 6] factors the language specification into separate static and dynamic parts, where each of these parts is represented as a distinct monad. For a language `L`, the compiler typically has type `(L->(Value D)S)` where monads `S` and `D` represent the static and dynamic parts of `L`. We use ML-style postfix type constructor application (e.g., "`Value D`" instead of "`D Value`"). The metacomputation `(S ∘ D)` is a `S`-computation which produces a `D`-computation, and although it is defined in terms of monads, it is generally not a monad itself.

A logical next step is to express metacomputation-style compilers within a staged functional language like MetaML. The effect of staging is visible in the type of the compiler, which is now `(L->(<Value D> S))`. Here the brackets `(< ... >)` indicate the MetaML *code* type constructor. Code values are first-class in MetaML and may be inspected as any other program data. An immediate advantage of using MetaML in this endeavor is that object code produced by the metacomputation-based compiler (i.e., code values of type `<Value D>`) are both observable as data and executable as MetaML programs.

When one applies a staged interpreter to a source program, a code valued object program is obtained which is a compiled version of the source program. That object program expresses instructions of an abstract machine, that is closer to the hardware than the source language. The dynamic monad plays an important role by encapsulating the abstract machine of the object code [21, 6, 14], and cleanly separating dynamic computation from static computation.

Although the guiding principle of previous work [7, 6] was to separate static from dynamic explicitly, we intentionally mix static with dynamic in one monad `M = S+D`[1] and use the MetaML code annotation `<...>` to distinguish static from

[1] Please pardon our abuse of language. The monads discussed here are constructed from monad transformers [14] and thus the additive notation is appropriate.

dynamic. Doing so allows us to compile using `S` features, execute using `D` features, and then re-compile, etc., all within the single monad `M`, and this interoperability between compilation and execution is *essential* to the current undertaking.

Because the static monad and the dynamic monad are the same, a very flexible approach to compilation is made possible. Dynamic computations can be performed at compile time (when information is available), and the dynamic computation can re-compile sub-components when needed. The use of a staged language makes the expression of such programs concise, high-level and natural.

When the abstract machine is able to perform staging, an object program can take advantage of this ability to adapt to runtime information. The following illustrates this matter concretely:

```
interpret        : Program -> input -> Value M
compile          : Program -> <input -> Value M> M
adaptcompile     : Program -> training -> <input -> Value M'> M'
```

where `M'` contains profiling information along with features from `M`. The function `interpret` is a monadic interpreter for the `Program` language, which when staged using metacomputation-style staging, produces `compile`. Enriching the monad `M` to `M'` yields a setting in which adaptable compilation becomes possible. Here, `training` is a set of training data to be used as an `input`. The compiled program is run on the training data collecting profile information, and is then recompiled to take advantage of the information learned in the training run. Using metacomputation and staging, this is easy to express, and can be added as a small delta to the program `compile`. It is the thesis of this paper that this technique can describe a wide variety of dynamically adaptable software.

2 Related Work

Traditionally constructed compilers [1, 15] have one key virtue: they can produce very high-quality target code through sophisticated program analysis techniques and aggressive code optimization. But their complicated structure makes them infamously difficult to prove correct. In contrast, semantics-based approaches to compilation [4, 7, 12, 28] are more amenable to formal proofs of correctness, but they fall short by comparison in the area of code optimization. The compiler writer appears to be in a dilemma, having to choose between performance (traditional compilers) and formal correctness (semantics-directed approaches). One purpose of this paper is to show that certain compiler optimizations may be easier to incorporate within a semantics-directed compilation scheme than within a traditional approach.

There is a synergy between the metacomputation-based approach to compilation and the staged language MetaML making it straightforward to implement profile-driven compiler optimizations. We found MetaML to be ideal because:

– Metacomputation-based language specifications, being explicitly staged, can be easily translated into the staged, functional language MetaML.

The first author developed metacomputation-based compilation as part of his thesis [6]. Previously, implementing the approach involved first translating the metacomputation-based specification into standard ML to ensure type correctness, and then translating it to Scheme by hand to apply partial evaluation. This involved annotating the resulting Scheme code (again by hand) with type information (and some black magic as well) to enable type-directed partial evaluation [3]. The result of partial evaluation is an observable Scheme program representing target code.

Given the same metacomputation-based specification written in standard ML, in contrast, achieving compilation is merely a matter of supplying the explicit staging annotations. This is orders of magnitude easier than the above process.

– MetaML's meta and object languages are the same making it easy to test metacomputation-based compilers written in MetaML and to test the object programs that they produce as well because both compiler and object code are MetaML programs.
– Metacomputation-based compiler specifications in MetaML are type-checked, thereby avoiding all sorts of latent errors.
– Profiling information may be added easily to metacomputation-based compiler specifications because metacomputations are structured by monad transformers [14].

Our framework for dynamically adaptable software may be viewed as an alternative to the "architectural composition" problem, where rich "architectures" are composed from modules or components. Such components need general interfaces which may be redundant or unnecessary in a larger composite architecture, and removing unnecessary component code is critical to making the approach practical for large systems. Architectural optimization usually consists of performing cross-component inlining to remove these unused interface components. The optimizer must treat the component code as data, observing their structure to determine which pieces to fuse. Treating "code as data" is integral to staged programming, and so in a dynamically adaptable software system as we have described, this "code as data" problem is handled at a much higher level of abstraction.

The remainder of this paper describes our case study in dynamically adaptable software and discusses its relevance as a model for such software. Section 3 defines a metacomputation-based compiler for a small imperative language with loops. Section 4 defines `adaptcompile` and discusses the minimal changes necessary to add dynamic reconfigurability to our reference compiler. Section 5 presents an example compilation. Section 6 outlines the general approach to adding a profile-driven optimization to a metacomputation-based compiler. Finally, Section 7 concludes with a discussion of how the techniques developed in this paper apply to adaptable software in general. In Appendix A, we have placed a short tutorial on the staging annotations and use of monads in MetaML. Appendix B gives an overview of the abstract machine and the monads used in the paper.

3 The Metacomputation-based Reference Compiler

```
               *********** Source Language ************
datatype Src = IntLit of int | Negate of Src | Add of Src*Src
             | Leq of Src*Src | ProgVar of Name | Assign of Name*Src
             | Seq of Src*Src | WhileDo of Src*Src;
datatype Program = Program of (Name list)*Src;

               *********** Static Operations **********
datatype Location = Loc of int | Reg;      rdEnv : Env M
datatype Env = env of Name -> Location;    rdAddr : Addr M
type Addr = int;                           inEnv : Env -> a M -> a M
type Label = int;                          inAddr : Addr -> a M -> a M
                                           newlabel : Label M

                ***** Abstract Machine Operations *****
datatype Value = code of <Value M> | Z of int | Void;
push : int -> Value M       branch : Label -> Label -> Value M
pop  : Value M              read,store      : Addr -> Value M
ADD,LEQ,NEG : Value M       pushReg,loadReg : Value M
jump : Label -> Value M     newSeg,endlabel : Label->Value M->Value M
```

Fig. 1. Source, Static & Abstract Machine Operations for the Reference Compiler

Figure 1 presents the source language for our compiler. We have deliberately kept this language quite simple, because we wish to keep our metacomputation-based compiler as simple as possible. `Src` is a simple imperative language with while loops. A program is represented as `(Program (globals,body))`, where `globals` are the globally-defined integer program variables with command `body` of type `Src`. `body` is either an assignment `(Assign(x,e))`, a loop `(WhileDo(b,c))`, or a sequence of such statements `(Seq(c1,c2))`. Variables are defined only in the declarations in the top-level `Program` phrase. `Src` has integer expressions `(IntLit i)`, `(Negate e)`, and `(Add(e1,e2))` representing integer constants, negation, and addition, respectively. `(ProgVar x)` is the program variable `x` when used as an expression. `Src` has only one boolean expression `(Leq(e1,e2))` standing for $(\leq)$.

In this Section we present a staged metacomputation-based compiler for `Program`. We use the combined monad approach discussed earlier. The monad `M` has two sets of operations, a set for static operations and another for dynamic operations. The static operations encapsulate the information needed at compile-time (e.g., mapping variable names to runtime locations), and the dynamic operations are the operations of the abstract machine.

The static operations of the monad `M` include mechanisms for manipulating environments, keeping track of free addresses, and generating fresh labels. They are given in the second part of Figure 1. The combinator `rdEnv` returns the current environment and `rdAddr` returns the current free address. For environment `r`, `(inEnv r x)` evaluates computation `x` in `r`. For address `a`, `(inAddr a x)` evaluates computation `x` with the "next free" address set to `a`. The combinator `newlabel` returns a fresh label.

The abstract machine targeted by our compiler is a simple stack machine with one general purpose register `Reg`. This is typical of target machines found in standard compiler textbooks [1, 15]. The key differences are that we give executable definitions for the abstract machine operations as monadic functions in MetaML, and we rely on MetaML's staging annotations to produce inspectable code. Our approach is similar to the way that type-directed partial evaluation was used in previous work [4, 8, 7, 6]. The operations of the targeted abstract machine are shown in the third part of Figure 1.

The MetaML definitions of these operations and of the monad `M` are given in Appendix B. The instruction `(push i)` pushes `i` onto the runtime stack and `pop` pops the runtime stack. The arithmetic instruction, `ADD`, pops the top two elements off of the stack, adds them, and pushes the result back on. The boolean test, `LEQ`, is defined similarly, although booleans are encoded as integer values (see the definitions of `encode`/`decode` in Appendix B.2 for further details). The jump `(jump L)` sends control to the code at label `L`, while `(branch L1 L2)` pops the stack and sends control to label `L1` (`L2`) if that value is true (false). Memory operation `(read a)` pushes the contents of address `a` onto the stack, while `(store a)` pops the top value from the stack and stores it in address `a`. Instructions `pushReg` and `loadReg` are similar to `read` and `store`, except that the source and target of the respective operation is the register `Reg`. Code store operation `(newSeg L pi)` defines a new code segment at `L`. Note that `(newSeg L pi)` does not execute `pi`, because `pi` is only executed if control is sent to label `L` with a `jump` or a `branch`. A *forward jump* is an instruction `(jump L)`, where the label `L` occurs after the `jump` instruction. Following Reynolds [20], we define a binding mechanism `(endlabel L pi)` defines a label `L` "at the end" of the code `pi`. One would pretty-print `(endlabel L pi)` as "`pi ; L:`". Forward jumps `(jump L)` within `(endlabel L pi)` simply branch to the "end" of `pi`.

Figure 2 presents the first compiler for the source language corresponding to the function `compile` from Section 1. `compile` is a MetaML function with type `(Program -> <int list -> Value M> M)`. Given a `Program`, it performs a computation which produces a piece of code with type `(int list -> Value M)`. The `(int list)` corresponds to the input to the program, which are the initial values for the global variables of the program.

Our reference compiler is similar to previous metacomputation-based compilers [7, 6] with two differences. Firstly, MetaML staging annotations appear explicitly in the definition of `compile` thereby eliminating the need for a partial evaluator to generate code (because `compile` produces "residual" code values `<...>` itself). The use of staging annotations in `compile` is evident in its type—note that the range of `compile` is a MetaML code value. A second difference between `compile` and previous metacomputation-based compilers [8, 7, 6] is that

```
(* ccsrc : Src -> <Value M> M *)
fun ccsrc e =
case e of
  (IntLit i) => Return M <push (~(lift i))>
| (Negate e) => Do M { pi <- (ccsrc e)  ; Return M <Do M { ~pi ; NEG }>}
| (Add (e1,e2)) => Do M { phi1 <- (ccsrc e1)
                        ; phi2 <- (ccsrc e2)
                        ; Return M <Do M { ~phi1 ; ~phi2 ; ADD }>}
| (Leq (e1,e2)) => Do M { phi1 <- (ccsrc e1)
                        ; phi2 <- (ccsrc e2)
                        ; Return M <Do M { ~phi1 ; ~phi2 ; LEQ }>}
| (Seq (c1,c2)) =>  Do M { phi1 <- ccsrc c1
                         ; phi2 <- ccsrc c2
                         ; Return M <Do M { ~phi1 ; ~phi2 }> }
| (WhileDo (b,c)) =>
    Do M { Lk <- newlabel ; Lc <- newlabel ; Lb <- newlabel
         ; phi_b <- ccsrc b
         ; phi_c <- ccsrc c
         ; Return M
               <endlabel ~(lift Lk)
                    (Do M { (newSeg ~(lift Lc)
                                 (Do M { ~phi_c
                                       ; jump ~(lift Lb) }))
                          ; (newSeg ~(lift Lb)
                                 (Do M { ~phi_b
                                       ; branch  ~(lift Lc) ~(lift Lk) }))
                          ; jump ~(lift Lb)   })>  }
| (ProgVar n) =>   Do M { (env rho) <- rdEnv
                        ; let val bnd_x = rho n
                          in case bnd_x of
                                Loc a => Return M <read ~(lift a)>
                              | Reg   => Return M <pushReg> end}
| (Assign (x,e)) =>
    Do M { phi_e <- (ccsrc e)
         ; env rho <- rdEnv
         ; let val bnd_x = (rho x)
           in case bnd_x of
                   Loc a => Return M <Do M { ~phi_e ; store ~(lift a) }>
                 | Reg   => Return M <Do M { ~phi_e ; loadReg}> end}

(* compile : Program -> <int list -> Value M> M *)
fun compile (Program (globals,main)) =
    Do M { fcode <- AllocVars (zip globals (upto (length globals - 1)))
                              (ccsrc main)
         ; Return M <fn input => ~(fcode <input>)>  };
```

Fig. 2. Compiler for Src and Program

`compile` is not explicitly derived from a denotational semantics for `Program`. We believe that this will make `compile` easier to understand.

The MetaML function `ccsrc` is a monadic version of what are sometimes called *semantic translation schemas* [1]. To see what is meant by this, consider first how the integer literals and negation in `Src` would be typically compiled into a stack language. Source expression `(IntLit i)` would be translated into the operation `(push i)`. Compiling `(Negate e)` would produce the stack code "`pi ; NEG`", where `pi` is the stack code translation of `e` and `NEG` is a command which pops the top value off of the stack and pushes its negation back on. It is assumed that executing `pi` results in the value for `e` being pushed on the stack.

Both of these translation schemas are easily made formal in MetaML. The code for `(IntLit i)` is just `<push (~(lift i))>`[2], and so `(ccsrc (IntLit i))` is just: `(Return M <push (~(lift i))>)`. If `(pi : <Value M>)` is the code for the expression `e`, then `<~pi ; NEG>` is the code for `(Negate e)`. Note that we had to splice `pi` into this code (i.e., use `~pi` instead of `pi`) because it is a MetaML code value. Now `(ccsrc (Negate e))` can be simply defined as:

```
Do M { pi <- (ccsrc e) ; Return M <Do M { ~pi ; NEG }>}
```

The other arithmetic and boolean expressions are compiled in a similar fashion.

Before Profiling Transformation: `x` and `y` are stored on stack

```
<(fn a =>
   Do M { push (nth (a,0))
        ; push (nth (a,1))
        ; endlabel 100
                   (Do M { newSeg 101
                                 (Do M { read 0 ; push 1 ; ADD ; store 0
                                       ; read 0 ; store 1 ; jump 102 })
                         ; newSeg 102 (Do M { read 0 ; push 3 ; LEQ ; branch 101 100 })
                         ; jump 102
                           })
        ; pop
        ; pop })>
```

After Profiling Transformation: `x` is now stored in `Reg`

```
<(fn a =>
   Do M' { push (nth (a,0))
         ; loadReg
         ; push (nth (a,1))
         ; endlabel 103
                   (Do M' { newSeg 104
                                 (Do M' { pushReg ; push 1 ; ADD ; loadReg
                                        ; pushReg ; store 0 ; jump 105 })
                          ; newSeg 105 (Do M' { pushReg ; push 3 ; LEQ ; branch 104 103 })
                          ; jump 105
                            })
         ; pop })>
```

Fig. 3. Compiling `(Program([x,y], while x<=3 { x:=x+1 ; y:=x }))`

[2] Here, the `~(lift i)` merely inlines the stage-0 value `i` into the stage-1 value `<push...>`.

Command sequencing `(Seq (c1,c2))` is a straightforward formalization of the translation:

$$(\texttt{Seq}(\texttt{c1},\texttt{c2})) \mapsto \text{“code for }\texttt{c1}\text{”}\texttt{;}\text{“code for }\texttt{c2}\text{”}$$

`(WhileDo (b,c))` is a formalization of the following translation schema:

```
(WhileDo(b,c)) ↦ For three new labels Lk, Lc, and Lb, emit:
                    Lc: "code for c" ; jump Lb
                    Lb: "code for b" ; branch Lc Lk
                        jump Lb
                    Lk:
```

The interesting cases for this compiler are the constructs involving use of program variables, because these may be stored in either stack locations or the register `Reg`. Specifically, these are `(ProgVar x)`, `(Assign(x,e))`, and `(Program (globals,main))`. `(ccsrc (ProgVar x))` must emit code either reading the current value of `x`. It checks whether `x` has been assigned a stack location `a` or the register `Reg`, and emits appropriate instruction `(read a)` or `loadReg`, respectively). Similarly, `(ccsrc (Assign(x,e)))` must determine where to store the top of stack (i.e., where `x` is kept). Initially, we assume that all program variables are stored on the stack, and `((compile (Program (globals,main))) input)` uses the function `AllocVars` to allocate stack locations for the global variables `globals`. The helper function `(var2loc x g)` allocates a new address `a`, and then runs the computation `g` in an extended environment in which `x` is bound to `a`:

```
(* var2loc : ['a]. string -> 'a M -> 'a M *)
fun var2loc x g =  Do M { a <- rdAddr
                        ; r <- rdEnv
                        ; initProfile x a
                        ; inAddr (a+1) (inEnv ((xEnv x (Loc a)) r) g) };

(* AllocVars : ['a]. (string * int) list -> (string * 'a ) Maybe ->
                              <Value M> M -> <int list> -> <Value M> M *)
fun AllocVars vars phi =
 case vars of
   [] => Do M { body <- phi ; Return M (fn input => body) }
 | (x,n)::xs =>
             var2loc x
                (Do M { fcode <- (AllocVars xs phi)
                      ; (Return M (fn input =>
                                   <Do M { push (nth(~input,~(lift n)))
                                           ; ~(fcode input)
                                           ; pop
                                           }>))});
```

In a call `(AllocVars vars phi)`, `(vars : (Name*int)list)` is a list of global variables paired with an index into the input list of initial values, while `(phi : <Value M>M)` is a code generator (i.e., an `M`-computation which produces code of type `<Value M>`). For each `(x,n)` in `vars`, the code produced by `phi` (i.e., `~(fcode input)` above) is enclosed within instructions which allocate and deallocate storage for `x`. Respectively, these instructions are a `push` of the `n`-th element of the input list to allocate and a `pop` to deallocate. Within the code produced by `phi`, `x` will be bound to the appropriate stack location.

A sample compilation is presented in the top half of Figure 3 (marked "Before"). In that figure, the accesses to `x` (which is bound to stack location `0`) are underlined.

3.1 Why Use MetaML `<...>` Instead of Concrete Syntax?

We could have defined an abstract syntax for the target stack language (e.g., `datatype Target = Push of int...`) and made `compile` a `Target`-valued computation. That is, `compile` could have been given type `Src -> Target M` and been defined as `compile (IntLit i) = Return M (Push i)`, etc. However, `Target` programs would not be immediately executable.

Because the target is an abstract machine parameterized by the monad `M`, simple changes to this machine can be used to collect profile information. We can extend the monad `M` to contain the material for both compiling `Src` programs, executing target machine programs, and profiling. This lays the foundation for dynamic adaptability.

4 Introducing Dynamic Profiling

What extensions are necessary to add dynamic profiling to the compiler in Figure 2? Having written `compile` in metacomputation-style, it is a simple matter to construct `adaptcompile`. We would like to write the following function and be done:

```
fun adaptcompile e training =
          Do M { pi <- compile e ; (run pi) training ; compile e };
```

As this point, however, the second call to `compile` would return the same code (modulo different `Labels`) as the first call (assuming `((run pi) training)` terminates). But remember, `(compile e)` is a monadic computation and can be affected by any states encapsulated in the monad and we run the first compiled version on the training data with the purpose of changing these states. To accomplish dynamic re-compilation, we must enrich the monad `M` to include a profile state and alter the abstract machine (`compile`) to make (or respond to) changes in the profile state.

We want to make minimal changes to `compile` to make `adaptcompile` work. Amazingly enough, the only changes needed are changes to the monad and the mechanism that `compile` uses to map variable names to locations. The exact changes are:

1. Enriched the monad `M` with a profile state (`type Profile = (string*int*int) list`). Call this new monad `M'`. Each element of this list is a profile of the form `(var,addr,count)`, where `var` is the program variable, `addr` is the address where it is stored, and `count` keeps track of the accesses to `var`. Define a function `(incUsageCount a : Value M')` which when executed will increment the count component of a profile `(var,a,count)`.

2. Include a call to `(incUsageCount a)` in the definitions of `(read a)` and `(store a)`.
3. Change the definition of `(AllocVars vars phi)` so that it picks a single program variable `x` with a maximal usage count from the `Profile` state and allocates the general-purpose register `Reg` for `x`.
4. Alter `compile` so that it computes the variable which should be stored in `Reg` (and call the resulting compiler `compile'`):

```
fun compile' (Program (globals,main)) =
  Do M' { maxP <- maxProfile (* compute maximally-used variable *)
        ; fcode <- AllocVars (zip globals (upto (length globals - 1)))
                             maxP
                             (ccsrc main)
        ; Return M' <fn input => ~(fcode <input>)>  };
```

After Step 3, `AllocVars looks like:`

```
fun useReg x maxP = case maxP of Nothing => false | Just (v,u) => v=x;

fun var2reg x g = Do M' { r <- rdEnv ; inEnv (xEnv x Reg r) g };

fun AllocVars vars maxP phi =
 case vars of
   [] => Do M' { body <- phi ; Return M' (fn input => body) }
 | (x,n)::xs =>
        if (useReg x maxP) then
            var2reg x
                (Do M' { fcode <- (AllocVars xs Nothing phi)
                       ; (Return M' (fn input =>
                                     <Do M' {  push (nth(~input,~(lift n)))
                                            ; loadReg
                                            ; ~(fcode input)}>))
                       })
        else
             var2loc x
                (Do M' { fcode <- (AllocVars xs Nothing phi)
                      ; (Return M' (fn input =>
                                     <Do M' { push (nth(~input,~(lift n)))
                                            ; ~(fcode input)
                                            ; pop
                                            }>))});
```

5 Example of Dynamic Recompilation

```
fun adaptcompile e training =
             Do M' { pi <- compile' e ; (run pi) training ; compile' e }
```

Fig. 4. Making `adaptcompile` from `compile` is a one-liner.

Figure 4 displays the compiler, `adaptcompile`, which dynamically recompiles its source program based on runtime profile information. Figure 3 shows "before and after" snapshots of compiling a program first without profiling and then with profiling information. Note first that the variable `x` is accessed most frequently in the sample program. In the "Before" snapshot, `x` is stored on the stack in location `0`, and accesses to `x` (which are underlined in both snapshots) are either `store` or `read` instructions. In the "After" snapshot, `x` is stored in the register `Reg`. First, the `0`-th member of the input list `a` is pushed onto the stack and loaded into `Reg`. Then, accesses to `x` are now performed with `pushReg` and `loadReg` instructions instead of `read` and `store` instructions. Finally, only one `pop` occurs at the end of the object code to deallocate `y`.

6 How to Add a Profile-driven Optimization to Any Metacomputation-based Compiler

Starting with compiler `(C:(L -> (input -> Value D)S))`:

1. Define `M` to be `(S + D)` incorporating all the operations of both monads. Use of monad transformers simplifies this step. Interpreting `C` within `M` has a different type, `(L -> (input -> Value M) M)`, but produces identical results.
2. Stage `C` by adding MetaML staging annotations to produce a new metacomputation-based compiler `(C' : L -> (<input -> Value M>) M)` for language `L` and monad `M`.
3. Fix the type of profile data `Profile` for the specific optimization. To add usage counts for program variables as we do in the present work, this is the type `(string*int*int) list`.
4. Form a new monad `M'` combining all of the features of the monad `M` with `Profile`. Because `M` is assumed to be constructed with monad transformers [14], this is a simple task. In our case study, `Profile` is added with a state monad transformer.
5. The metacomputation-based compiler `(C' : L->(<input -> Value M'>) M')` behaves just as `(C' : L -> (<input -> Value M>) M)`.
6. Alter the target language combinators to use `Profile` information. In our case, we change only `(read a)` and `(store a)`.
7. Construct a new version of `adaptC` that observes profile data (when it exists) and compiles accordingly.

7 Conclusions and Future Work

How compiler optimizations are to be performed within a semantics-based approach to compilation has frequently been unclear. In this paper we have outlined a general technique for adding a whole class of compiler optimizations—those that are profile-driven—within a particular form of semantics-directed compiler. Adding profiling to the metacomputation-based reference compiler shown here was quite simple, mainly because the static and dynamic aspects of a metacomputation-based compiler can easily be combined. We did not include performance measures, as our goal was the development of a structure encapsulating dynamic adaptability rather than demonstrating the usefulness of any particular compiler optimization.

For the purpose of specifying and proving correctness, there is a significant advantage to keeping separate static and dynamic monads, `S` and `D`, instead of combining them into one monad (as we did here with the monad `M`). But this creates a complication in implementing the systems as we have in this paper in that a `D` computation is not immediately executable within a `S` computation. One approach is to introduce a "lift" morphism, which embeds a `D` computation into a `S` computation. In this setting, `adaptcompile` would look like:

```
compile : Program -> <input -> Value D> S;
lift : a D -> a S;
adaptcompile : Program -> training -> <input -> Value D> S
fun adaptcompile e training =
 Do S { pi <- compile e ; lift ((run pi) training) ; compile e }
```

We believe this technique is an instance of a general structure for modeling dynamically adaptable programs. If a staged programming language is used to implement both the compiler for a language, and the run-time system of the language, one essentially gets run-time re-compilation for free. This allows the expression of a whole range of dynamically adaptable behaviors at a very high level of abstraction.

Staged metacomputation-based compilers provide a high-level interface for describing adaptable systems. Suppose a system which evolves periodically, and for which significant speed-ups are possible if the source could be re-compiled to take advantage of configuration changes.

The system might be a network driver, and its configuration information may be the capacities of switches, lines, and the network topography. Or, the system might be an operating system service, and its configuration information may be the number, locations, and types of disk drives. Imagine that significant performance improvements can be made by compiling the code to take advantage of new configuration information. Several choices are possible:

1. One can pre-compile a number of anticipated configurations and dynamically switch between these when configurations change. The disadvantages of this mechanism are that it might be hard or impossible to anticipate all such changes.
2. Write an interpreter over the configuration states (with the consequent loss of performance that interpretation implies).
3. Bring down the system when configurations change for re-compilation. Obviously this involves human interference and significant loss of service.

The great thing about staged computation is that we can express at a high-level a single solution which incorporates the best aspects of all three of these solutions all at a modest cost. We take the second approach, choosing an interpreter because of its great flexibility. Staging it produces a compiler giving us the benefits of the first option. But there is no need to anticipate changes as new versions can be produced on demand. Merging the static and dynamic computations (and thereby enabling staging at the abstract machine level) captures in a high-level and concise manner both the kinds of policies implicit in the third option and the procedures that would be used to implement them. The approach outlined here brings together compilation, scripting languages, execution and monitoring abilities into one unified framework. For example, the high-level description of the above scenario is concisely described by:

```
adaptcompile prog config = Do M { code <- compile e config
                                ; handle (run code)
                                         (fn c => adaptcompile prog c) }
```

Here the monad has an exception mechanism which is raised when changes in the configuration occur. The function `(handle body handler)` runs `body`, which may raise an exception when the configuration state changes. At that point, `h`andler continues execution until the next safe state is reached. Then, control is then passed to `handler` which, when supplied with the new configuration state, re-compiles and continues. Note that the configuration information `config` may contain a label specifying the entry (or re-entry) point in the code.

Acknowledgments

The work described here was supported by NSF Grant CDA-9703218, the M.J. Murdock Charitable Trust and the Department of Defense.

The authors would also like to thank Sam Kamin and Emir Pasalic for discussions of these ideas.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Reading, Mass.: Addison-Wesley, 1985.
2. C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 145–156, New York, NY, USA, 1996. ACM Press.
3. O. Danvy. Type-directed partial evaluation. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 242–257, New York, NY, USA, 1996. ACM Press.
4. O. Danvy and R. Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. *Lecture Notes in Computer Science*, 1140:182–209, 1996.

5. R. Glück and J. Jørgensen. Efficient multi-level generating extensions for program specialization. In S. D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 1995.
6. W. Harrison. *Modular Compilers and Their Correctness Proofs.* PhD thesis, University of Illinois at Urbana-Champaign, 2001.
7. W. Harrison and S. Kamin. Metacomputation-based compiler architecture. In *Mathematics of Program Construction – MPC200, Proc. 5th International Conference on the Mathematics of Program Construction, Ponte de Lima, Portugal*, volume 1837 of *Lecture Notes in Computer Science*, pages 213–229. Springer-Verlag, 2000.
8. W. L. Harrison and S. N. Kamin. Modular compilers based on monad transformers. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 122–131. IEEE Computer Society Press, 1998.
9. P. Hudak, S. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partian, and J. Peterson. Report on the programming language haskell, version 1.2. *Sigplan*, 27(5), May 1992. Hudak, Wadler, Arvind, Boutel, Fairbairn, Fasel, Hughes, Johnsson, Kieburtz, Nikhil, Peyton Jones, Reeve, Wise, Young; Version 1.0: Functional Programming (Languages?) and Computer Architecture 89, pp123, 1989.
10. M. P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 52–64, New York, NY, USA, June 1993. ACM Press.
11. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).
12. P. Lee. *Realistic Compiler Generation.* Foundations of Computing Series. MIT Press, 1989.
13. M. Leone and P. Lee. A declarative approach to run-time code generation. In *Workshop on Compiler Support for System Software (WCSSS)*, Feb. 1996.
14. S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press.
15. S. S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA, 1997.
16. M. Odersky and K. Läufer. Putting type annotations to work. In ACM, editor, *Conference record of POPL '96, 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: St. Petersburg Beach, Florida, 21–24 January 1996*, pages 54–67, New York, NY, USA, 1996. ACM Press.
17. J. Peterson, K. Hammond, et al. Report on the programming language haskell, a non-strict purely-functional programming language, version 1.3. Technical report, Yale University, May 1996.
18. Calton Pu and Jonathan Walpole. A study of dynamic optimization techniques: Lessons and directions in kernel design. Technical Report OGI-CSE-93-007, Oregon Graduate Institute of Science and Technology, 1993.
19. C. Pu, H. Massalin, and J. Ioannidis. The synthesis kernel. *Usenix Journal, Computing Systems*, 1(1):11, Winter 1988.

20. J. C. Reynolds. The essence of algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, Amsterdam, 1981. North-Holland.
21. T. Sheard, Z. El-Abidine Benaissa, and E. Pasalic. DSL implementation using staging and monads. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 81–94, Berkeley, CA, Oct. 3–5 1999. USENIX Association.
22. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics.* MIT Press, Cambridge, Massachusetts, 1977.
23. W. Taha. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.
24. W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97)*, volume 32, 12 of *ACM SIGPLAN Notices*, pages 203–217, New York, June 12–13 1997. ACM Press.
25. P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.
26. P. Wadler. The essence of functional programming. In ACM, editor, *Conference record of POPL '92, 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: Albuquerque, New Mexico, January 1992*, pages 1–15.
27. P. Wadler. Monads for functional programming. *Lecture Notes in Computer Science*, 925:24–52, 1995.
28. M. Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, July 1982.

A MetaML Tutorial

MetaML is almost a conservative extension of Standard ML. Its extensions include four staging annotations. To delay an expression until the next stage one places it between meta-brackets. Thus the expression `<23>` (pronounced "bracket 23") has type `<int>` (pronounced "code of int"). The annotation, ~*e* splices the deferred expression obtained by evaluating *e* into the body of a surrounding Bracketed expression; and `run` *e* evaluates *e* to obtain a deferred expression, and then evaluates this deferred expression. It is important to note that ~*e* is only legal within lexically enclosing Brackets. We illustrate the important features of the staging annotations in the short MetaML sessions below.

```
-| val z = 3+4;
val z = 7 : int
```

Users access MetaML through a *read-type-eval-print* top-level. The declaration for `z` is read, type-checked to see that it has a consistent type (`int` here), evaluated (to `7`), and then both its value and type are printed.

```
-| val quad = ( 3+4,  <3+4>,    lift (3+4), <z>  );
val quad =    ( 7,    <3 %+ 4>, <7>,        <%z> ) :
              ( int * <int> *   <int> *     <int>)
```

The declaration for `quad` contrasts normal evaluation with the three ways objects of type code can be constructed. Placing brackets around an expression (`<3+4>`) defers the computation of `3+4` to the next stage, returning a piece of code. Lifting an expression (`lift (3+4)`) evaluates that expression (to `7` here) and then lifts the value to a piece of code that when evaluated returns the same value. Brackets around a free variable (`<z>`) creates a new constant piece of code with the value of the variable. Such constants print with a `%` sign to indicate they are constants. We call this *lexical-capture* of free variables. Because in MetaML operators (such as `+` and `*`) are also identifiers, free occurrences of operators in constructed code often appear with `%` in front of them.

```
-| fun inc x = <1 + ~x>;
val inc = Fn  : ['a].<int> -> <int>
```

The declaration of the function `inc` illustrates that larger pieces of code can be constructed from smaller ones by using the escape annotation. Bracketed expressions can be viewed as *frozen*, i.e. evaluation does not apply under brackets. However, is it often convenient to allow some reduction steps inside a large frozen expression while it is being constructed, by "splicing" in a previously constructed piece of code. MetaML allows one to *escape* from a frozen expression by prefixing a sub-expression within it with the tilde (~) character. Escape must only appear inside brackets.

```
-| val six = inc <5>;
val six =  <1 %+ 5> : <int>
```

In the declaration for `six`, the function increment is applied to the piece of code `<5>` constructing the new piece of code `<1 %+ 5>`.

```
-| run six;
val it = 6 : int
```

Running a piece of code, strips away the enclosing brackets, and evaluates the expression inside. To give a brief feel for how MetaML is used to construct larger pieces of code at runtime consider:

```
-| fun mult x n = if n=0 then <1> else < ~x * ~(mult x (n-1)) >;
val mult = fn  : <int> -> int  -> <int>

-| val cube = <fn y => ~(mult <y> 3)>;
val cube = <fn a => a * (a * (a * 1))> : <int  -> int>

-| fun exponent n = <fn y => ~(mult <y> n)>;
val exponent = fn  : int  -> <int  -> int>
```

The function `mult`, given an integer piece of code `x` and an integer `n`, produces a piece of code that is an `n`-way product of `x`. This can be used to construct the code of a function that performs the `cube` operation, or generalized to a generator for producing an exponentiation function from a given exponent `n`. Note how the looping overhead has been removed from the generated code. This is the purpose of program staging and it can be highly effective as discussed elsewhere [2, 5, 13, 18, 24].

A.1 Monads in MetaML

We assume the reader has a working knowledge of monads [25, 27]. We use the *unit* and *bind* formulation of monads [26]. In MetaML a monad is a data structure encapsulating a type constructor M and the *unit* and *bind* functions.

```
datatype ('M : * -> * ) Monad = Mon of
   (['a]. 'a -> 'a 'M) *                          (* unit function *)
   (['a,'b]. 'a 'M -> ('a -> 'b 'M) -> 'b M);  (* bind function *)
```

This definition uses ML's postfix notation for type application, and two non-standard extensions to ML. First, it declares that the argument `('M : * -> * )` of the type constructor `Monad` is itself a unary type constructor [10]. We say that `'M` has *kind*: `* -> *`. Second, it declares that the arguments to the constructor `Mon` must be polymorphic functions [16]. The type variables in brackets, e.g. `['a,'b]`, are universally quantified. Because of the explicit type annotations in the `datatype` definitions the effect of these extensions on the Hindley-Milner type inference system is well known and poses no problems for the MetaML type inference engine.

In MetaML, `Monad` is a first-class, although *pre-defined* or *built-in* type. In particular, there are two syntactic forms which are aware of the `Monad` datatype: `Do` and `Return`. `Do` and `Return` are MetaML's syntactic interface to the *unit* and *bind* of a monad. We have modeled them after the do-notation of Haskell [9, 17]. An important difference is that MetaML's `Do` and `Return` are both parameterized by an expression of type `'M Monad`. `Do` and `Return` are syntactic sugar for the following:

```
(* Syntactic Sugar                           Derived Form       *)
   Do (Mon(unit,bind)) { x <- e; f }   =   bind e (fn x => f)
   Return (Mon(unit,bind)) e           =   unit e
```

In addition the syntactic sugar of the `Do` allows a sequence of `x`$_i$ `<- e`$_i$ forms, and defines this as a nested sequence of `Do`'s. For example:

```
Do m { x1 <- e1; x2 <- e2 ; x3 <- e3 ; e4 }    =
   Do m { x1 <- e1; Do m { x2 <- e2 ; Do m { x3 <- e3 ; e4 }}}
```

Users may freely construct their own monads, though they should be very careful that their instantiation meets the monad axioms. The monad axioms, expressed in MetaML's `Do` and `Return` notation are:

```
Do {x <- Return e ; z}          = z[e/x]
Do {x <- m ; Return x}          = m
Do {x <- Do { y <- a ; b } ; c} = Do {y' <- a ; Do { x <- b[y'/y] ; c }}
                                = Do {y' <- a ; x <- b[y'/y] ; c}
```

B Executable Definition of Target Code

This appendix describes the executable specification of the abstract machine which is the target of the compiler in Figure 2.

We assume that the reader is familiar with monads [26, 27, 25, 14]. The monads used in this paper are constructed from monad transformers [14]. A monad transformer creates a new monad from an existing monad and adds new data and operations to manipulate that data. The following table summarizes the monad transformers used in this paper:

M.T.	Associated Operation(s)	Meaning
$\mathcal{T}_{\mathsf{Env}}$ `T`	`rdT : T M`	current `T` environment
	`inT: T->a M->a M`	`(inT t x)` evals `x` in `T`-env `t`
$\mathcal{T}_{\mathsf{St}}$ `S`	`updateS : (S->S)->S M`	`(updateS f)` applies `f` to current `S` state, returning the resulting state
	`getS : S M`	`getS = (updateS id)` returns current `S` state
$\mathcal{T}_{\mathsf{CPS}}$ `A`	`CallCC:(a->A M)->A M`	`(CallCC f)` passes `f` the current contin.

There are four monads used (sometimes implicitly) in this paper. They are:

$$\mathtt{S} = \mathcal{T}_{\mathsf{Env}}\ \mathtt{Env}\ (\mathcal{T}_{\mathsf{Env}}\ \mathtt{Addr}\ (\mathcal{T}_{\mathsf{St}}\ \mathtt{Label}\ \mathtt{Id}))$$
$$\mathtt{D} = \mathcal{T}_{\mathsf{CPS}}\ \mathtt{Value}\ (\mathcal{T}_{\mathsf{St}}\ \mathtt{Sto}\ (\mathcal{T}_{\mathsf{St}}\ \mathtt{Code}\ \mathtt{Id}))$$
$$\mathtt{M} = \mathcal{T}_{\mathsf{Env}}\ \mathtt{Env}\ (\mathcal{T}_{\mathsf{Env}}\ \mathtt{Addr}\ (\mathcal{T}_{\mathsf{CPS}}\ \mathtt{Value}\ (\mathcal{T}_{\mathsf{St}}\ \mathtt{Sto}\ (\mathcal{T}_{\mathsf{St}}\ \mathtt{Label}\ (\mathcal{T}_{\mathsf{St}}\ \mathtt{Code}\ \mathtt{Id})))))$$
$$\mathtt{M}' = \mathcal{T}_{\mathsf{Env}}\ \mathtt{Env}\ (\mathcal{T}_{\mathsf{Env}}\ \mathtt{Addr}\ (\mathcal{T}_{\mathsf{CPS}}\ \mathtt{Value}\ (\mathcal{T}_{\mathsf{St}}\ \mathtt{Sto}\ (\mathcal{T}_{\mathsf{St}}\ \mathtt{Label}\ (\mathcal{T}_{\mathsf{St}}\ \mathtt{Code}\ (\mathcal{T}_{\mathsf{St}}\ \mathtt{Profile}\ \mathtt{Id}))))))$$

`S` and `D` are the static and dynamic monads, respectively, from the original metacomputation-based compiler for `Src`. One can recover this compiler from Figure 2 by replacing each occurrence of `M` occurring within `<...>` as `D` and all other occurrences with `S`. Erasing the staging annotations would then recover the original compiler. `M` arises as the combination `S+D` in step 1 on page 12. Finally, `M'` is the monad `S+D+Profile` from step 3 on page 12.

For the monad `M'` above:

```
Env = string -> Location (where Location = Loc of int | Reg)
Addr = Label = int
Value = code of <Value M'> | Z of int | Void
Sto = Addr*int*(Addr->Value)
Code = segm of Label -> Value M'
Profile = (string*int*int) list
```

In `(sp,Reg,sigma):Sto`, `sp` is the current stack pointer, `Reg` is the current contents of the general register, and `sigma` is the memory map (only integer values are stored in `sigma`). The code store `Code` is used to store continuations. Modeling jumps with a stored continuations is a common technique from denotational semantics [22].

B.1 Static Operations of `M`

The only static operation of the monad `M` not defined directly by the monad transformers is `newlabel`:

```
val newlabel = Do M { L <- updateLabel (fn l => l)
                    ; _ <- updateLabel (fn l => l+1) ; Return M L};
```

B.2 The Abstract Machine

All of the following are defined in terms of the operations provided by the above monad transformers. Except for `read` and `store`, their definitions in `M'` are identical.

Stack operations:

```
fun tweek l v sigma = fn l' => if l = l' then v else sigma l';
fun update f = Do M { _ <- updateSto f ; Return M Void};
fun writeLoc a i = update (fn (sp,reg,sigma)  => (sp,reg,tweek a i sigma));
fun rdLoc a = Do M { (sp,reg,sigma) <- getSto  ; Return M (Z (sigma a)) };
fun setSP sp = update (fn (_,reg,sigma)  => (sp,reg,sigma));
fun setReg reg = update (fn (sp,_,sigma)  => (sp,reg,sigma));
val SP = Do M { (sp,reg,sigma) <- getSto ; Return M sp };
val Reg = Do M { (sp,reg,sigma) <- getSto ; Return M reg };
fun push i = Do M { sp <- SP ; writeLoc sp i ; setSP (sp+1) };
val pop = Do M { sp <- SP ; setSP (sp-1) ; rdLoc (sp-1) };
val pushReg = Do M { reg <- Reg ; push reg };
val loadReg = Do M { (Z i) <- pop ; setReg i };
fun read a = Do M { Z i <- rdLoc a ; push i };
fun store a = Do M { (Z i) <- pop ; writeLoc a i };
```

Profiling operations:

```
fun incUsage a pl =  case pl of
   ((n,a',i)::prf) => if (a = a') then ((n,a',i+1)::prf)
                                  else  ((n,a',i)::incUsage a prf)
 | [] => [];

fun incUsageCount a = updateProfile (incUsage a);
fun read a = Do M' { Z i <- rdLoc a ; incUsageCount a ; push i };
fun store a = Do M' { (Z i) <- pop ; incUsageCount a ; writeLoc a i };
```

Arithmetic/Boolean operations:

```
val NEG = Do M { (Z i) <- pop ; push (neg i) };
val ADD = Do M { (Z i) <- pop ; (Z j) <- pop ; push (i + j) };
fun encode tf = if tf then 888 else 999;
fun decode i = (i=888);
val LEQ = Do M { (Z v1) <- pop ; (Z v2) <- pop ; push (encode (v2<=v1)) };
```

Control-flow operations:

```
fun rdSeg L = Do M { (segm Pi) <- getCode ; Pi L };
fun newSeg L x = updateCode (fn (segm Pi) => segm (tweek L x Pi));
fun endlabel L pi = CallCC (fn k => Do M { _ <- newSeg L (k Void) ; pi });
fun jump L = CallCC (fn _ => rdSeg L);
fun branch Lt Lf = Do M { (Z bv) <- pop
                        ; if (decode bv) then (jump Lt) else (jump Lf) };
```