Compilation as Metacomputation: Binding Time Separation in Modular Compilers (*Extended Abstract*)

William L. Harrison Samuel N. Kamin

Department of Computer Science
University of Illinois, Urbana-Champaign
Urbana, Illinois 61801-2987
{harrison,kamin}@cs.uiuc.edu

Abstract

This paper presents a modular and extensible style of language specification based on metacomputations. This style uses two monads to factor the static and dynamic parts of the specification, thereby staging the specification and achieving strong binding-time separation. Because metacomputations are defined in terms of monads, they can be constructed modularly and extensibly using monad transformers. A number of language constructs are specified: expressions, control-flow, imperative features, block structure, and higher-order functions and recursive bindings. Metacomputation-style specification lends itself to semantics-directed compilation, which we demonstrate by creating a modular compiler for a higher-order, imperative, Algol-like language.

Keywords: Compilers, Partial Evaluation, Semantics-Based Compilation, Programming Language Semantics, Monads, Monad Transformers, Pass Separation.

1 Introduction

Metacomputations—computations that produce computations—arise naturally in the compilation of programs. Figure 1 illustrates this idea. The source language program `s` is taken as input by the compiler, which produces a target language program `t`. So, compiling `s` produces another computation—namely, the computation of `t`. Observe that there are two entirely distinct notions of computation here: the compilation of `s` and the execution of `t`. The reader will recognize this distinction as the classic separation of static from dynamic. Thus, *staging* is an instance of metacomputation.

The main contribution of this paper is a modular and extensible method of staging denotational specifications based on metacomputations, formalized via monads[7, 13, 16, 23]. A style of

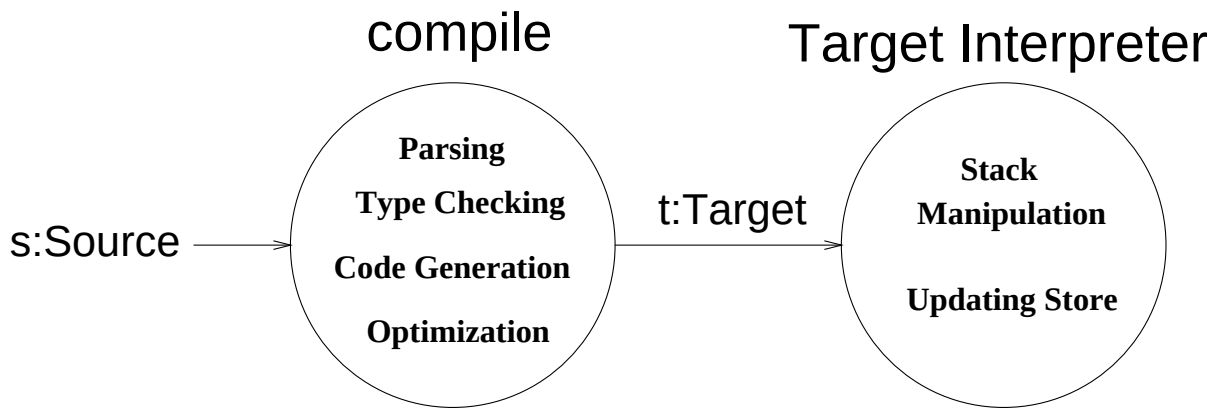

Figure 1: Handwritten compiler as metacomputation

language specification based on metacomputation is proposed in which the static and dynamic parts of a language specification are factored into distinct monads.

We believe this style of language specification may have many uses, but in this paper we concentrate on one: modular compilation. *Modular compilers* are compilers built from building blocks that represent *language features* rather than *compilation phases*, as illustrated in Figure 2. Espinosa [7] and Liang & Hudak [13] showed how to construct modular *interpreters* using the notion of *monads* [7, 13, 16, 23] — or, more precisely, monad *transformers*.

The current authors built on those ideas to produce modular *compilers* in [8]. However, there the notion of *staging*, though conceptually at the heart of the approach, was not *explicit* in the compiler building blocks we constructed. As in traditional monadic semantics, the monadic structure was useful in creating the domains, but those domains, once constructed, were "monolithic;" that is, they gave no indication of which parts were for dynamic aspects of the computation and which for static aspects. The result was awkwardness in communicating between these aspects of the domain, which meant that "gluing together" compiler blocks was sometimes delicate.

Indeed, metacomputation is *purposely* avoided in [7, 13, 8]. A key aspect of that work is that monad transformers are used to create the single monad used to interpret or compile the language. The problem that inspired it was that monads don't compose nicely. Given monads M and M', the composed monad $M \circ M'$ — corresponding to an M-computation that produces an M' computation — usually does not produce the "right" monolithic domain. However, there may exist monad transformers T_M and $T_{M'}$ such that $T_M\ id = M$ and $T_{M'}\ id = M'$, where $(T_M \circ T_{M'})id$ does give the "right" domain. The difference between composing monads and composing monad transformers is what makes these approaches work — monad transformers are a way to *avoid* metacomputation.

In this paper, we show that, for some purposes, metacomputation may be exactly what you

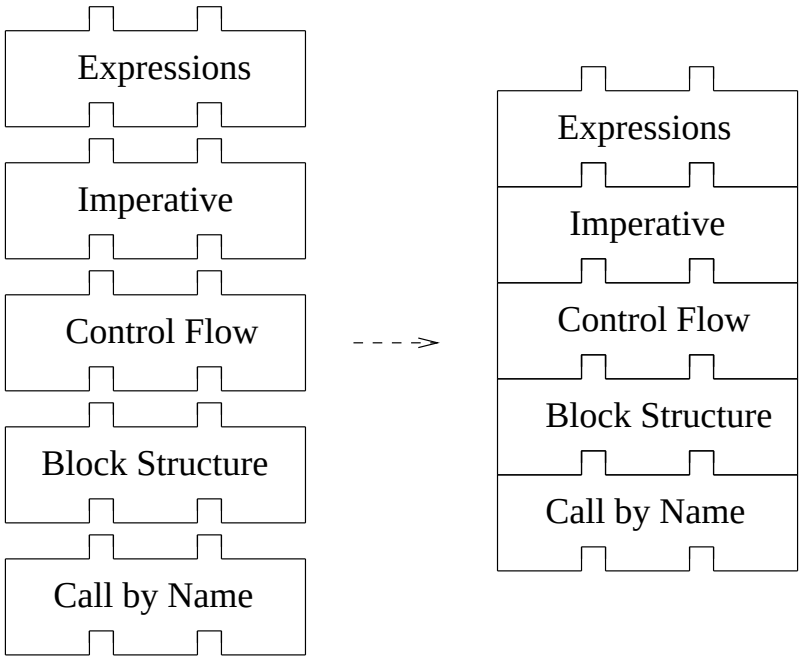

Figure 2: Modular Compilers: Existing compiler building blocks combine to make new compiler

want: *Defining a compiler block via the metacomputation of* two *monads gives an effective representation of staging.* We are not advocating abandoning monad transformers: the two monads can be constructed using them, with the attendant advantages of that approach. We are simply saying that having two monads — what might be called the *static* and *dynamic* monads — and composing them seems to give the "right" domain for modular compilation.

This paper also includes two substantive improvements over [8] that, though less essential, also help to make the approach more practical. First, instead of writing all specifications in continuation-passing style, here we write in direct style, invoking the CPS monad transformer only when needed; this naturally simplifies many of the equations. Second, we include some building blocks not given in [8], specifically optimized expressions and procedures. *(These will be included in the full paper.)* Together these give us the entire language of Reynolds [21] with essentially the same code generated there (but with jumps and labels, avoiding the potential for infinite programs).

The next section reviews the most relevant related work. In Section 3, we review the theory of monads and monad transformers and their use in language specification. Section 4 presents a case study in metacomputation-style language specification; its subsections present metacomputation-style specifications for expressions, control flow, block structure, and imperative features, respectively. (*In the full paper, we will include the specifications for optimized expressions and procedures.*) Section 5 shows how to combine these compiler building blocks into a compiler for

the combined language, and presents a compiler and an example compilation. Finally, Section 6 summarizes this work and outlines future research.

2 Related work

Espinosa [7] and Hudak, Liang, and Jones [13] use monad transformers to create modular, extensible interpreters. Liang [12, 14] addresses the question of whether compilers can be developed similarly, but since he does not compile to machine language, many of the issues we confront—especially staging—do not arise.

A syntactic form of metacomputation can be found in the two-level λ-calculus of Nielson[19]. Two-level λ-calculus contains two distinct λ-calculi—representing the static and dynamic *levels*. Expressions of mixed level, then, have strongly separated binding times by definition. Nielson[18] applies two-level λ-calculus to code generation for a typed λ-calculus, and Nielson[19] presents an algorithm for static analysis of a typed λ-calculus which converts one-level specifications into two-level specifications. Mogensen[15] generalizes this algorithm to handle variables of mixed binding times. The present work offers a semantic alternative to the two-level λ-calculus. We formalize distinct levels (in the sense of Nielson[19]) as distinct monads, and the resulting specifications have all of the traditional advantages of monadic specifications (reusability, extensibility, and modularity). While our binding time analysis is not automatic as in [19, 15], we consider a far wider range of programming language features than they do.

Danvy and Vestergaard [5] show how to produce code that "looks like" machine language, by expressing the source language semantics in terms of machine language-like combinators (e.g., "popblock", "push"). When the interpreter is closed over these combinators, partial evaluation of this closed term with respect to a program produces a completely *dynamic* term, composed of a sequence of combinators, looking very much like machine language. This approach is key to making the monadic structure useful for compilation.

Reynolds' [21] demonstration of how to produce efficient code in a compiler derived from the functor category semantics of an Algol-like language was an original inspiration for this study. Our compiler for that language (*presented in the full paper*) improves on Reynolds's in two ways: it is monad-structured—that is, built from interchangeable parts—and it includes jumps and labels where Reynolds simply allowed code duplication and infinite programs.

3 Monads and Monad Transformers

In this section, we review the theory of monads [16, 23] and monad transformers [7, 13]. Readers familiar with these topics may skip the section.

A *monad* is a type constructor M together with a pair of functions (obeying certain algebraic laws that we omit here):

$$\star_{\mathsf{M}} : \mathsf{M}\tau \to (\tau \to \mathsf{M}\tau') \to \mathsf{M}\tau'$$
$$\mathbf{unit}_{\mathsf{M}} : \tau \to \mathsf{M}\tau$$

A value of type $\mathsf{M}\tau$ is called a *τ-computation*, the idea being that it yields a value of type τ while also performing some other computation. The $\star_M$ operation generalizes function application in that it determines how the computations associated with monadic values are combined. $\mathbf{unit}_\mathsf{M}$ defines how a τ value can be regarded as a τ-computation; it is usually a trivial computation.

To see how monads are used, suppose we wish to define a language of integer expressions containing constants and addition. The standard definition might be:

$$[\![e_1{+}e_2]\!] \;=\; [\![e_1]\!] + [\![e_1]\!]$$

where $[\![-]\!] : \mathit{Expression} \to \mathit{int}$. However, this definition is inflexible; if expressions needed to look at a store, or could generate errors, or had some other feature not planned on, the equation would need to be changed.

Monads can provide this needed flexibility. To start, we rephrase the definition of $[\![-]\!]$ in monadic form (using infix bind $\star$, as is traditional) so that $[\![-]\!]$ has type $\mathit{Expression} \to \mathsf{M}\,\mathit{int}$:

$$[\![e_1{+}e_2]\!] \;=\; [\![e_1]\!] \star (\lambda x.[\![e_2]\!] \star (\lambda y.\mathtt{add}(x,y)))$$

We must define an operation `add` of type $\mathit{int} \times \mathit{int} \to \mathsf{M}\,\mathit{int}$.

The beauty of the monadic form is that the meaning of $[\![-]\!]$ can be reinterpreted in a variety of monads. Monadic semantics separate the *description* of a language from its *denotation*. In this sense, it is similar to *action semantics*[17] and *high-level semantics*[11].

The simplest monad is the identity monad, shown in Figure 3. Given the identity monad, we can define `add` as ordinary addition. $[\![-]\!]$ would have type $\mathit{Expression} \to \mathit{int}$.

Perhaps the best known monad is the state monad, which represents the notion of a computation as something that modifies a store:

$$\begin{array}{l}
\mathsf{M}_{St}\tau \;=\; \mathit{Sto} \to \tau \times \mathit{Sto} \\
x \star f \;=\; \lambda\sigma.\ \mathbf{let}\ (x',\sigma') = x\sigma\ \mathbf{in}\ fx'\sigma' \\
\mathbf{u}nit\, v \;=\; \lambda\sigma.(v,\sigma) \\
\mathtt{add}\,(x,y) \;=\; \lambda\sigma.(x+y,\sigma)
\end{array}$$

The $\star$ operation handles the bookkeeping of "threading" the store through the computation. Now, $[\![-]\!]$ has type $\mathit{Expression} \to \mathit{Sto} \to \mathit{int} \times \mathit{Sto}$. This might be an appropriate meaning for addition in an imperative language. To define operations that actually have side effects, we can define a function:

$$\begin{array}{ll}
\mathtt{updateSto} & : (\mathit{Sto} \to \mathit{Sto}) \to \mathsf{M}_{St}\mathtt{void} \\
 & : f \mapsto\ \lambda\sigma.(\bullet, f\sigma) \\
\mathtt{getSto} & : \mathsf{M}_{St}\mathit{Sto} \\
 & : \lambda\sigma.(\sigma,\sigma)
\end{array}$$

`updateSto` applies a function to the store and returns a useless value (we assume a degenerate type `void` having a single element, which we denote $\bullet$). `getSto` returns the store.

Identity Monad Id:

$\mathsf{Id}\,\tau = \tau$
$\mathbf{unit}_{\mathsf{Id}}\,x = x$
$x \star_{\mathsf{Id}}\ f = f\,x$

Environment Monad Transformer $\mathcal{T}_{\mathsf{Env}}$:

$\mathsf{M}'\tau = \mathcal{T}_{\mathsf{Env}}\,Env\,\mathsf{M}\,\tau = Env \to \mathsf{M}\tau$
$\mathbf{unit}_{\mathsf{M}'}\,x = \lambda\rho : Env.\,\mathbf{unit}_{\mathsf{M}}\,x$
$x \star_{\mathsf{M}'}\ f = \lambda\rho : Env.\,(x\,\rho) \star_{\mathsf{M}}\ (\lambda a.f\,a\,\rho)$
$lift_{\mathsf{M}\tau\to\mathsf{M}'\tau}\,x = \lambda\rho : Env.\,x$
$\mathtt{rdEnv} : \mathsf{M}'Env = \lambda\rho : Env.\,\mathbf{unit}_{\mathsf{M}}\rho$
$\mathtt{inEnv}(\rho : Env,\, x : \mathsf{M}'\tau) = \lambda\_.\,(x\,\rho) : \mathsf{M}'\tau$

State Monad Transformer $\mathcal{T}_{\mathsf{St}}$:

$\mathsf{M}'\tau = \mathcal{T}_{\mathsf{St}}\,store\,\mathsf{M}\,\tau = store \to \mathsf{M}(\tau \times store)$
$\mathbf{unit}_{\mathsf{M}'}\,x = \lambda\sigma : store.\,\mathbf{unit}_{\mathsf{M}}(x,\sigma)$
$x \star_{\mathsf{M}'}\ f = \lambda\sigma_0 : store.\,(x\,\sigma_0) \star_{\mathsf{M}}\ (\lambda(a,\sigma_1).f\,a\,\sigma_1)$
$lift_{\mathsf{M}\tau\to\mathsf{M}'\tau}\,x = \lambda\sigma.\,x \star_{\mathsf{M}} \lambda y.\,\mathbf{unit}_{\mathsf{M}}(y,\sigma)$
$\mathtt{update}(\Delta : store \to store) = \lambda\sigma.\,\mathbf{unit}_{\mathsf{M}}(\bullet, \Delta\,\sigma)$
$\mathtt{getStore} = \lambda\sigma.\,\mathbf{unit}_{\mathsf{M}}(\sigma,\sigma)$

CPS Monad Transformer $\mathcal{T}_{\mathsf{CPS}}$:

$\mathsf{M}'\tau = \mathcal{T}_{\mathsf{CPS}}\,ans\,\mathsf{M}\,\tau = (\tau \to \mathsf{M}\,ans) \to \mathsf{M}\,ans$
$\mathbf{unit}_{\mathsf{M}'}\,x = \lambda\kappa.\,\kappa\,x$
$x \star_{\mathsf{M}'}\ f = \lambda\kappa.\,x(\lambda a.f\,a\,\kappa)$
$lift_{\mathsf{M}\tau\to\mathsf{M}'\tau}\,x = \star_{\mathsf{M}}$

Figure 3: The Identity Monad, and Environment, State, and CPS Monad Transformers

Now, suppose a computation can cause side effects on two separate stores. One could define a new "double-state" monad M_{2St}:

$$\mathsf{M}_{2St}\tau \;=\; \mathit{Sto} \times \mathit{Sto} \to \tau \times \mathit{Sto} \times \mathit{Sto}$$

that would thread the two states through the computation, with separate `updateSto` and `getSto` operations for each copy of *Sto*. One might expect to get $\mathsf{M}_{2St}\tau$ by applying the ordinary state monad twice. Unfortunately, $\mathsf{M}_{St}(\mathsf{M}_{St}\tau)$ and $\mathsf{M}_{2St}\tau$ are very different types. This points to a difficulty with monads: they do not compose in this simple manner.

The key contribution of the work [7, 13] on *monad transformers* is to solve this composition problem. When applied to a monad M, a monad transformer $\mathcal{T}$ creates a new monad M'. For example, the state monad transformer, $\mathcal{T}_{St}\,\mathit{store}$, is shown in Figure 3. (Here, the *store* is a type argument, which can be replaced by any value which is to be "threaded" through the computation.) Note that $\mathcal{T}_{St}\,\mathit{Sto}\,\mathsf{Id}$ is identical to the state monad, but here we get a useful notion of composition: $\mathcal{T}_{St}\,\mathit{Sto}\,(\mathcal{T}_{St}\,\mathit{Sto}\,\mathsf{Id})$ is equivalent to the two-state monad $\mathsf{M}_{2St}\tau$. The state monad transformer also provides `updateSto` and `getSto` operations appropriate to the newly-created monad. When composing $\mathcal{T}_{St}\,\mathit{Sto}$ with itself, as above, the operations on the "inner" state need to be *lifted* through the outer state monad; this is the main technical issue in [7, 13].

In our work in [8], we found it convenient to factor the state monad into two parts: the state proper and the address allocator. This was really a "staging transformation," with the state monad representing dynamic computation and the address allocator static computation, but, as mentioned earlier, it led to significant complications. In the current paper, we are separating these parts more completely, by viewing compilation as metacomputation.

4 A Case Study in Metacomputation-style Staging: Modular Compilation for the While Language

In this section, we present several compiler building blocks. In section 5, they will be combined to create a compiler. For the first two of these blocks, we also give monolithic versions, drawn from [8], to illustrate why metacomputation is helpful.

4.1 Integer Expressions Compiler Building Block

Consider the standard monadic-style specification of negation[7, 13, 23] displayed in Figure 4. To use this as a compiler specification for negation, we need to make a more implementation-oriented version, which might be defined informally as:

$$\begin{array}{l}
[\![-t]\!] \;= \\
\quad [\![t]\!] \;\star_E\; \lambda i. \\
\quad \text{"Store } i \text{ at } a \text{ and return contents of } a\text{"} \;\star_E\; \lambda v. \\
\qquad \mathbf{unitE}\;(-v)
\end{array}$$

Standard:

$\mathsf{Exec} = \mathsf{Id}$

$[\![-t]\!] : \mathsf{Exec}(int) =$
$\quad [\![t]\!] \star_E \ \lambda i.$
$\quad\quad \mathbf{unitE}\ (-i)$

Implementation-oriented/Monolithic:

$\mathsf{Exec} = \mathcal{T}_{\mathsf{Env}}\ Addr\ (\mathcal{T}_{\mathsf{St}}\ Sto\ \mathsf{Id})$
$Addr = int, Sto = Addr \to int$
$\mathtt{Thread}(i : int, a : Addr) =$
$\quad \mathtt{updateSto}[a \mapsto i] \ \star_E \ \lambda\_.\mathtt{rdloc}(a)$
$\mathtt{rdloc}(a) = \mathtt{getSto} \ \star_E \ \lambda\sigma.\mathbf{unitE}(\sigma a)$

$[\![-t]\!] : \mathsf{Exec}(int) =$
$\quad [\![t]\!] \star_E \ \lambda i.$
$\quad \mathtt{rdAddr} \star_E \ \lambda a.$
$\quad \mathtt{inAddr}\ (a+1)$
$\quad\quad (\mathtt{Thread}(i, a) \ \star_E \ \lambda v.\mathbf{unitE}\ (-v))$

Metacomputation:

$\mathsf{Exec} = \mathcal{T}_{\mathsf{St}}\ Sto\ \mathsf{Id}$
$\mathsf{Comp} = \mathcal{T}_{\mathsf{Env}}\ Addr\ \mathsf{Id}$

$\mathcal{C}[\![-t]\!] : \mathsf{Comp}(\mathsf{Exec}(int)) =$
$\quad \mathtt{rdAddr} \star_C \ \lambda a.$
$\quad \mathtt{inAddr}\ (a+1)$
$\quad\quad (\mathcal{C}[\![t]\!] \ \star_C \ \lambda\phi_t : \mathsf{Exec}(int).$
$\quad\quad\quad \mathbf{unitC} \boxed{\left(\begin{array}{l} \phi_t \ \star_E \ \lambda i. \\ \mathtt{Thread}(i, a) \ \star_E \ \lambda v. \\ \quad \mathbf{unitE}(-v) \end{array}\right)}$

Figure 4: Negation, 3 ways

Let us assume that this is written in terms of a monad Exec with bind and unit operations $\star_E$ and **unitE**. Observe that this implementation-oriented definition calculates the same value as the standard definition, but it stores the intermediate value i as well. But where do addresses and storage come from? In [8], we added them to the Exec monad using monad transformers[7, 13] as in the "Implementation-oriented" specification in Figure 4. In that definition, `rdAddr` reads the current top of stack address a, `inAddr` increments the top of stack, and `Thread` stores i at a. The monad (Exec) is used to construct the domain containing both static and dynamic data.

In the "metacomputation"-style specification, we use two monads, Comp, to encapsulate the static data, and Exec to encapsulate the dynamic data. The meaning of the phrase is a metacomputation—the Comp monad produces a computation of the Exec monad. Clear separation of binding times is thus achieved. (In our examples, we have set the dynamic parts of the computation in a box for emphasis.)

Figure 5 displays the specification for addition, which is similar to negation. Multiplication and subtraction are defined analogously.

$$
\begin{array}{l}
\mathcal{C}[\![e_1 + e_2]\!] : \mathsf{Comp}(\mathsf{Exec}\, int) = \\
\quad \texttt{rdAddr} \star_C \lambda a. \\
\quad \mathcal{C}[\![e_1]\!] \star_C \lambda \phi_{e_1}. \\
\quad \texttt{inAddr}\ (a+2) \\
\quad\ \mathcal{C}[\![e_2]\!] \star_C \lambda \phi_{e_2}. \\
\qquad \mathbf{unitC} \left(\begin{array}{l}
\phi_{e_1} \star_E \lambda i : int. \\
\phi_{e_2} \star_E \lambda j : int. \\
\texttt{Thread}(i, a) \star_E \lambda v_1. \\
\texttt{Thread}(j, (a+1)) \star_E \lambda v_2. \\
\quad \mathbf{unitE}(v_1 + v_2)
\end{array} \right)
\end{array}
$$

Figure 5: Specification for Addition

4.2 Control-flow Compiler Building Block

We now present an example where separating binding times in specifications with metacomputations has a very significant advantage over the monolithic approach. Consider the three definitions of the conditional **if-then** statement in Figure 6. The first is a dual continuation "control-flow" semantics, found commonly in compilers[2]. If B is true, then the first continuation, $[\![c]\!] \star_E \kappa$, is executed, otherwise c is skipped and just κ is executed. A more implementation-oriented (informal) specification might be:

$$
\begin{array}{l}
[\![\mathbf{if}\ b\ \mathbf{then}\ c]\!] = \\
\quad [\![b]\!] \star_E \lambda B. \\
\quad \text{“get two new labels } L_c, L_\kappa\text{”} \star_E \lambda \langle L_c, L_\kappa \rangle. \\
\quad \texttt{callcc}\ (\lambda \kappa. \\
\qquad \text{“store } \kappa \text{ at } L_\kappa\text{, then } ([\![c]\!] \star_E (\text{“jump to } L_\kappa\text{”}))\text{ at } L_c\text{”} \star_E \lambda\_. \\
\qquad B\langle \text{“jump to } L_c\text{”}, \text{“jump to } L_\kappa\text{”} \rangle)
\end{array}
$$

To formalize this specification, we use a technique from denotational semantics for modeling jumps. We introduce a continuation store, *Code*, and a label state *Label*. A jump to label L simply invokes the continuation stored at L. The second definition in Figure 6 presents an implementation-oriented specification of **if-then** in monolithic style (that is, where *Code* and *Label* are both added to Exec). Again, this represents our approach in [8].

One very subtle problem remains: what is "`newSegment`"? One's first impulse is to define it as a simple update to the *Code* store (i.e., $\texttt{updateCode}[L_\kappa \mapsto \kappa]$), but here is where the monolithic approach greatly complicates matters. Because the monolithic specification mixes static and

Control-Flow:

$\mathsf{Exec} = \mathcal{T}_{\mathsf{CPS}}\ \mathtt{void}\ \mathsf{Id}$
$Bool = \forall\alpha.\alpha \times \alpha \to \alpha$

$[\![\mathbf{if}\ b\ \mathbf{then}\ c]\!]\mathsf{Exec}(\mathtt{void}) =$
$\quad [\![b]\!]\ \star_E\ \lambda B : Bool.$
$\quad \mathtt{callcc}\ (\lambda\kappa.$
$\quad\quad B\langle [\![c]\!]\ \star_E\ \kappa, \kappa\rangle)$

Implementation-oriented/Monolithic:

$\mathsf{Exec} = \mathcal{T}_{\mathsf{CPS}}\ \mathtt{void}\,(\mathcal{T}_{\mathsf{St}}\ Label\ (\mathcal{T}_{\mathsf{St}}\ Code\ \mathsf{Id}))$
$Label = int, Code = \mathtt{void} \to \mathsf{Exec}\ \mathtt{void}$
$\mathtt{jump}\ L = \mathtt{getCode}\ \star_E\ (\lambda\Pi : Code.\Pi\ L)$
$\mathtt{newlabel} : \mathsf{Exec}(Label) =$
$\quad \mathtt{getLabel}\ \star_E\ \lambda l : Label.$
$\quad \mathtt{updateLabel}[L \mapsto L+1]\ \star_E\ \lambda\_.$
$\quad \mathbf{unitE}(l)$

$[\![\mathbf{if}\ b\ \mathbf{then}\ c]\!] : \mathsf{Exec}(\mathtt{void}) =$
$\quad [\![b]\!]\ \star_E\ \lambda B : Bool.$
$\quad \mathtt{newlabel}\ \star_E\ \lambda L_\kappa.$
$\quad \mathtt{newlabel}\ \star_E\ \lambda L_c.$
$\quad \mathtt{callcc}\ (\lambda\kappa.$
$\quad\quad \mathtt{newSegment}(L_\kappa, \kappa)\ \star_E\ \lambda\_.$
$\quad\quad \mathtt{newSegment}(L_c, [\![c]\!]\ \star_E\ (\mathtt{jump}\ L_\kappa))\ \star_E$
$\quad\quad\quad B\langle \mathtt{jump}\ L_c, \mathtt{jump}\ L_\kappa\rangle)$

Metacomputation:

$\mathsf{Exec} = \mathcal{T}_{\mathsf{CPS}}\ \mathtt{void}\ (\mathcal{T}_{\mathsf{St}}\ Code\ \mathsf{Id})$
$\mathsf{Comp} = \mathcal{T}_{\mathsf{St}}\ Label\ \mathsf{Id}$

$\mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c]\!] : \mathsf{Comp}(\mathsf{Exec}\ \mathtt{void}) =$
$\quad \mathcal{C}[\![b]\!]\ \star_C\ \lambda\phi_B.$
$\quad \mathcal{C}[\![c]\!]\ \star_C\ \lambda\phi_c.$
$\quad \mathtt{newlabel}\ \star_C\ \lambda L_c.$
$\quad \mathtt{newlabel}\ \star_C\ \lambda L_\kappa.$
$\quad \mathbf{unitC}\ \left(\begin{array}{l} \phi_B\ \star_E\ \lambda B : Bool. \\ \mathtt{callcc}\ (\lambda\kappa. \\ \quad \mathtt{updateCode}[L_\kappa \mapsto \kappa]\ \star_E\ \lambda\_. \\ \quad \mathtt{updateCode}[L_c \mapsto \phi_c\ \star_E\ (\mathtt{jump}\ L_\kappa)]\ \star_E \\ \quad\quad B\langle \mathtt{jump}\ L_c, \mathtt{jump}\ L_\kappa\rangle) \end{array}\right)$

Figure 6: **if-then**: 3 ways

$$
\begin{array}{l}
\mathcal{C}[\![e_1 \leq e_2]\!] : \mathsf{Comp}(\mathsf{Exec}\, Bool) = \\
\quad \texttt{rdAddr} \star_C \lambda a. \\
\quad \mathcal{C}[\![e_1]\!] \star_C \lambda \phi_{e_1}. \\
\quad \texttt{inAddr}\ (a+2) \\
\quad\ \mathcal{C}[\![e_2]\!] \star_C \lambda \phi_{e_2}. \\
\qquad \mathbf{unitC} \left(\begin{array}{l}
\phi_{e_1} \star_E \lambda i : int. \\
\phi_{e_2} \star_E \lambda j : int. \\
\texttt{Thread}(i, a) \star_E \lambda v_1. \\
\texttt{Thread}(j, (a+1)) \star_E \lambda v_2. \\
\quad \mathbf{unitE} \left(\begin{array}{l}
\lambda \langle \kappa_T, \kappa_F \rangle. \\
\quad ((v_1 \leq v_2) \ \rightarrow\ \kappa_T, \kappa_F)
\end{array} \right)
\end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] : \mathsf{Comp}(\mathsf{Exec}\ \texttt{void}) = \\
\quad \mathcal{C}[\![b]\!] \star_C \lambda \phi_B. \\
\quad \mathcal{C}[\![c]\!] \star_C \lambda \phi_c. \\
\quad \texttt{newlabel} \star_C \lambda L_{test}. \\
\quad \texttt{newlabel} \star_C \lambda L_c. \\
\quad \texttt{newlabel} \star_C \lambda L_\kappa. \\
\qquad \mathbf{unitC} \left(\begin{array}{l}
\texttt{callcc}\ \lambda \kappa. \\
\quad \phi_B \star_E \lambda B : Bool. \\
\quad \texttt{updateCode}[L_\kappa \mapsto \kappa] \star_E \lambda \_. \\
\quad \texttt{updateCode}[L_c \mapsto \phi_c \star_E (\texttt{jump}\ L_{test})] \star_E \\
\quad \texttt{updateCode}[L_{test} \mapsto \phi_B \star_E \lambda B.((B \langle \texttt{jump}\ L_c, \texttt{jump}\ L_\kappa \rangle \bullet)] \star_E \\
\quad\ \texttt{jump}\ L_{test}
\end{array} \right)
\end{array}
$$

Figure 7: Specification for $\leq$ and **while**

dynamic computation, the continuation κ may contain both kinds of computation. But because it is *stored* and not *executed*, κ will not have access to the current label count and any other static data necessary for proper staging. Therefore, `newSegment` must explicitly pass the current label count and any other static intermediate data structure to the continuation it stores[1].

The last specification in Figure 6 defines **if-then** as a metacomputation and is much simpler than the monolithic-style specification. Observe that Exec does not include the *Label* store, and so the continuation κ now includes only dynamic computations. Therefore, there is no need to pass in the label count to κ, and so, κ may simply be stored in *Code*.

[1] A full description of `newSegment` is found in [8].

$$\mathsf{Exec} = \mathsf{Id}$$
$$\mathsf{Comp} = \mathcal{T}_{\mathsf{Env}}\, Env\, (\mathcal{T}_{\mathsf{Env}}\, Addr\, \mathsf{Id})$$
$$\mathtt{set}\, a = \lambda v.\mathtt{updateSto}(a \mapsto v)$$
$$\mathtt{get}\, a = \mathtt{getSto}\ \star_E\ \lambda\sigma.\mathbf{unitE}(\sigma\, a)$$

$$\begin{array}{l}\mathcal{C}[\![\mathbf{new}\ x\ \mathbf{in}\ c]\!] : \mathsf{Comp}(\mathsf{Exec}\ \mathtt{void}) = \\ \quad \mathtt{rdAddr}\ \star_C\ \lambda a. \\ \quad \mathtt{inAddr}\,(a+1) \\ \quad\ \mathtt{rdEnv}\ \star_C\ \lambda\rho. \\ \quad\ \mathtt{inEnv}\,(\rho[x \mapsto \mathbf{unitC}\ \langle \mathtt{set}\, a, \mathtt{get}\, a\rangle])\ \mathcal{C}[\![c]\!]\end{array}$$

Figure 8: Compiler Building Block for Block Structure

$$\mathsf{Exec} = \mathcal{T}_{\mathsf{St}}\, Sto\, \mathsf{Id},\ \mathsf{Comp} = \mathcal{T}_{\mathsf{Env}}\, Env\, \mathsf{Id}$$
$$\begin{array}{l}\mathcal{C}[\![c_1;c_2]\!] : \mathsf{Comp}(\mathsf{Exec}\ \mathtt{void}) = \\ \quad \mathcal{C}[\![c_1]\!]\ \star_C\ \lambda\phi_{c_1}. \\ \quad \mathcal{C}[\![c_2]\!]\ \star_C\ \lambda\phi_{c_2}. \\ \quad\ \mathbf{unitC}\ \boxed{(\phi_{c_1}\ \star_E\ \lambda\_.\phi_{c_2})}\end{array}$$

$$\begin{array}{l}\mathcal{C}[\![x := t]\!] : \mathsf{Comp}(\mathsf{Exec}\ \mathtt{void}) = \\ \quad \mathtt{rdEnv}\ \star_C\ \lambda\rho. \\ \quad (\rho\, x)\ \star_C\ \lambda\langle acc, \_\rangle. \\ \quad \mathcal{C}[\![t]\!]\ \star_C\ \lambda\phi_t. \\ \quad\ \mathbf{unitC}\ \boxed{(\phi_t\ \star_E\ \lambda i : int.(acc\ i))}\end{array}$$

Figure 9: Compiler Building Block for Imperative Features

Figure 7 contains the specifications for $\leq$ and **while**, which are very similar to the specifications of addition and **if-then**, respectively, that we have seen already.

4.3 Block Structure Compiler Building Block

The block structure language includes **new** x **in** c, which declares a new program variable x in c. The compiler building block for this language appears in Figure 9. The static part of this specification allocates a free stack location a, and the program variable x is bound to an accepter-expresser pair[20] in the current environment ρ. In an *accepter-expresser* pair $\langle acc, exp\rangle$, acc accepts an integer value and sets the value of its variable to the value, and the expresser exp simply returns the current value of the variable. `set` and `get` set and return the contents of location a, respectively. c is then compiled in the updated environment and larger stack $(a+1)$.

4.4 Imperative Features Compiler Building Block

The simple imperative language includes assignment (:=) and sequencing (;). The compiler building block for this language appears in Figure 9. For sequencing, the static part of the specification compiles c_1 and c_2 in succession, while the dynamic (boxed) part runs them in succession. For assignment, the static part of the specification retrieves the accepter[20] acc for program variable x from the current environment ρ and compiles t, while the dynamic part calculates the value of t and passes it to acc.

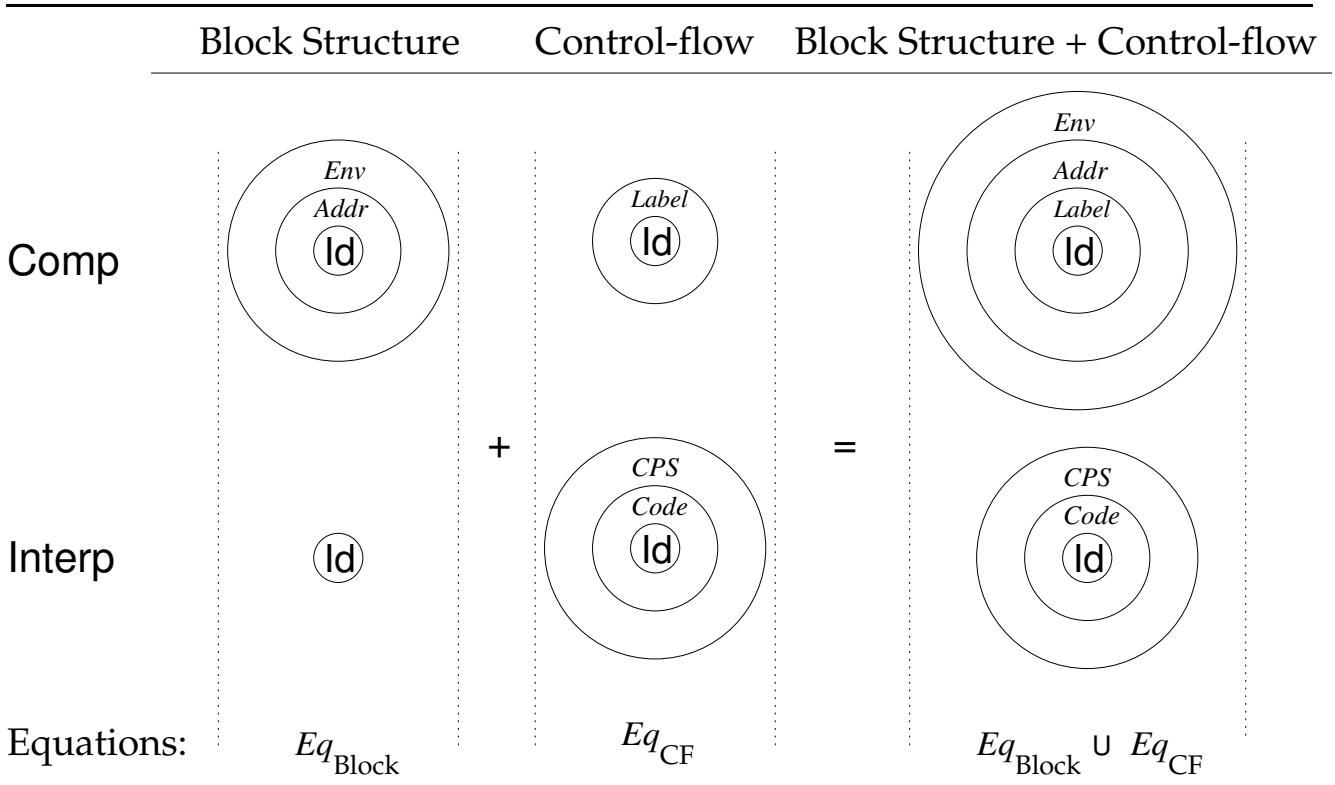

Figure 10: Combining Compiler Building Blocks

4.5 Optimized Expression Compiler Building Block

Reynolds [21] gives a more efficient translation of arithmetic expressions than we have given in our expression building block. This building block reproduces Reynolds's code.

(This section is omitted from the abstract.)

4.6 Procedures Compiler Building Block

(This section is omitted from the abstract.)

5 Combining Compiler Building Blocks

Figure 10 illustrates the process of combining the compiler building blocks for the block structure and control-flow languages. The process is nothing more than applying the appropriate monad transformers to create the Comp and Exec monads for the combined language. Recall that for the block structure language:

$$\mathsf{Comp} = \mathcal{T}_{\mathsf{Env}}\, Env\, (\mathcal{T}_{\mathsf{Env}}\, Addr\, \mathsf{Id}), \text{ and } \mathsf{Exec} = \mathsf{Id}$$

Compiler:

$\mathsf{Exec} = \mathcal{T}_{\mathsf{CPS}}\ \texttt{void}\ (\mathcal{T}_{\mathsf{St}}\ \mathit{Code}\ (\mathcal{T}_{\mathsf{St}}\ \mathit{Sto}\ \mathsf{Id}))$, $\mathsf{Comp} = \mathcal{T}_{\mathsf{Env}}\ \mathit{Env}\ (\mathcal{T}_{\mathsf{Env}}\ \mathit{Addr}\ (\mathcal{T}_{\mathsf{St}}\ \mathit{Label}\ \mathsf{Id}))$

Language = Expressions $\cup$ Imperative $\cup$ Control-flow $\cup$ Block structure $\cup$ Booleans

Equations = $Eq_{\text{Expressions}} \cup Eq_{\text{Imperative}} \cup Eq_{\text{Control-flow}} \cup Eq_{\text{Block structure}} \cup Eq_{\text{Booleans}}$

Source Code:

```
new x in
    new y in
        x := 5; y := 1;
        while (1 ≤ x) do
            y := y*x; x := x-1;
```

Target Code:

```
        0 := 5;                  2:  2 := [1];                  3:  halt;
        1 := 1;                      3 := [0];
        jump 1;                      1 := [2] * [3];
                                     2 := [0];
1:      2 := 1;                      3 := 1;
        3 := [0];                    0 := [2] - [3]
        BRLEQ [2] [3] 2 3;           jump 1;
```

Figure 11: Compiler for While language and example compilation

For the control flow language:

$$\mathsf{Comp} = \mathcal{T}_{\mathsf{St}}\ \mathit{Label}\ \mathsf{Id}, \text{ and } \mathsf{Exec} = \mathcal{T}_{\mathsf{CPS}}\ \texttt{void}\ (\mathcal{T}_{\mathsf{St}}\ \mathit{Code}\ (\mathcal{T}_{\mathsf{St}}\ \mathit{Sto}\ \mathsf{Id}))$$

To combine the compiler building blocks for these languages, one simply combines the respective monad transformers:

$$\mathsf{Comp} = \mathcal{T}_{\mathsf{Env}}\ \mathit{Env}\ (\mathcal{T}_{\mathsf{Env}}\ \mathit{Addr}\ (\mathcal{T}_{\mathsf{St}}\ \mathit{Label}\ \mathsf{Id})), \text{ and } \mathsf{Exec} = \mathcal{T}_{\mathsf{CPS}}\ \texttt{void}\ (\mathcal{T}_{\mathsf{St}}\ \mathit{Code}\ (\mathcal{T}_{\mathsf{St}}\ \mathit{Sto}\ \mathsf{Id}))$$

Now, the specifications for both of the smaller languages, Eq_{Block} and Eq_{CF}, apply for the "larger" Comp and Exec monads, and so we have the compiler for the combined language is specified by $Eq_{Block} \mathsf{U} Eq_{CF}$.

Figure 11 contains the compiler for the while language, and an example program and its pretty-printed compiled version. All that was necessary was to combine the compiler building blocks developed in this section combined as discussed in Subsection 5.

6 Conclusions and Future Work

This paper presents a modular and extensible style of language specification based on metacomputation. This style uses two monads to factor the static and dynamic parts of the specification,

thereby staging the specification and achieving strong binding-time separation. Because metacomputations are defined in terms of monads, they can be constructed modularly and extensibly using monad transformers. We exploit this fact to create modular *compilers*.

Future work focuses on two areas: specifying other language constructs like objects, classes, and exceptions; and exploring the impact of the metacomputation-style on compiler correctness. A question of particular interest is constructing modular correctness proofs of modular compilers.

Acknowledgements

The authors would like to thank Olivier Danvy for implementing his type-directed partial evaluator in SCM for us. Uday Reddy offered many helpful suggestions that led to significant improvements in the presentation.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, 1986.

[2] A. Appel, *Modern Compiler Implementation in ML,* Cambridge University Press, New York, 1998.

[3] A. Appel, *Compiling with Continuations,* Cambridge University Press, New York, 1992.

[4] O. Danvy, "Type-Directed Partial Evaluation," *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1996.

[5] O. Danvy and R. Vestergaard, "Semantics-Based Compiling: A Case Study in Type-Directed Partial Evaluation," *Eighth International Symposium on Programming Language Implementation and Logic Programming*, 1996, pages 182-497.

[6] R. Davies and F. Pfenning, "A Modal Analysis of Staged Computation," *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1996.

[7] D. Espinosa, "Semantic Lego," Doctoral Dissertation, Columbia University, 1995.

[8] W. Harrison and S. Kamin, "Modular Compilers Based on Monad Transformers," *Proceedings of the IEEE International Conference on Programming Languages*, 1998, pages 122-131.

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

[10] U. Jorring and W. Scherlis, "Compilers and Staging Transformations," *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1986.

[11] P. Lee, *Realistic Compiler Generation,* MIT Press, 1989.

[12] S. Liang, "A Modular Semantics for Compiler Generation," *Yale University Department of Computer Science Technical Report TR-1067*, February 1995.

[13] S. Liang, P. Hudak, and M. Jones, Monad Transformers and Modular Interpreters. *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1995.

[14] S. Liang, "Modular Monadic Semantics and Compilation," Doctoral Thesis, Yale University, 1997.

[15] T. Mogensen. "Separating Binding Times in Language Specifications," *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pp 12-25, 1989.

[16] E. Moggi, "Notions of Computation and Monads," *Information and Computation 93(1),* pp. 55-92, 1991.

[17] P. Mosses, *Action Semantics,* Cambridge University Press, 1992.

[18] H. Nielson and F. Nielson, "Code Generation from two-level denotational metalanguages," in *Programs as Data Objects*, Lecture Notes in Computer Science **217** (Springer, Berlin, 1986).

[19] H. Nielson and F. Nielson, "Automatic Binding Time Analysis for a Typed λ-calculus," *Science of Computer Programming* 10, 2 (April 1988), pp 139-176.

[20] J. Reynolds. "The Essence of Algol," *Algorithmic Languages, Proceedings of the International Symposium on Algorithmic Languages*, pp. 345-372, 1981.

[21] J. Reynolds, "Using Functor Categories to Generate Intermediate Code," *Proceedings of the ACM Conference on the Principles of Programming Languages*, pages 25–36, 1995.

[22] J. E. Stoy, *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory,* MIT Press, 1977.

[23] P. Wadler, "The essence of functional programming," *Proceedings of the ACM Conference on the Principles of Programming Languages*, pages 1–14, 1992.

[24] M. Wand, "Deriving Target Code as a Representation of Continuation Semantics," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, pp. 496-517, 1982.