Metacomputation-based Compiler Architecture

William L. Harrison and Samuel N. Kamin

University of Illinois at Urbana-Champaign, Urbana IL 61801, USA
harrison@cs.uiuc.edu,kamin@cs.uiuc.edu,
WWW home page: http://www-sal.cs.uiuc.edu/~{harrison,kamin}

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, and block structure. Metacomputation-style specification lends itself to semantics-directed compilation, which we demonstrate by creating a modular compiler for a block-structured, imperative while 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 contributions of this paper are: (1) *Compiler architecture based on metacomputations:* Metacomputation-based compiler architecture yields substantially simpler language definitions than in [8], while still retaining its modular "mix and match" approach to compiler construction. Combining the metacomputation-based "reusable compiler building blocks" is also much simpler than combining those in [8] (as is proving their correctness). (2) *A modular and extensible method of staging denotational specifications based on metacomputations:* A style of language specification based on metacomputation is proposed in which the static and dynamic parts of a language specification are factored into distinct monads[7, 14, 18, 27]. (3) *Direct-style specifications:* instead of writing all specifications in continuation-passing style, here we write in direct style,

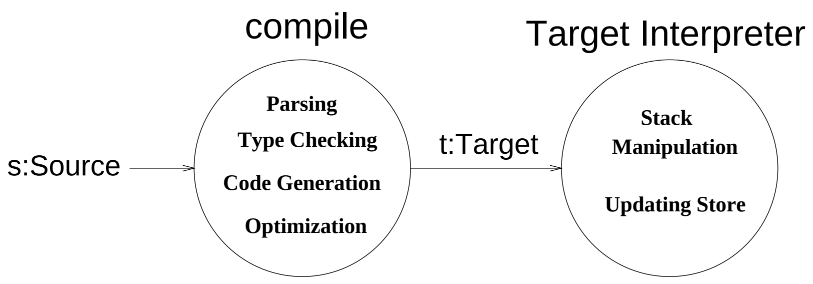

Fig. 1. Handwritten compiler as metacomputation

invoking the CPS monad transformer only when needed. This naturally simplifies many of the equations, and although less essential than (1) and (2), it also helps to make the approach more practical.

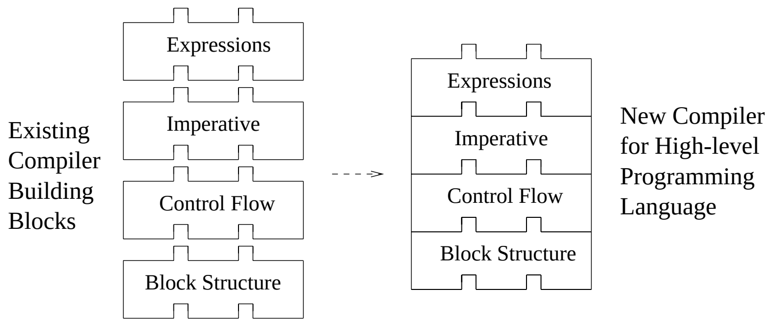

Fig. 2. Modular compilers constructed with existing compiler building blocks

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, & Jones [14] showed how to construct modular interpreters using the notion of monads[7, 14, 18, 27] — 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. However, metacomputation-based compiler architecture *completely* alleviates this awkwardness, so that combining compiler blocks is simply a matter of applying the appropriate monad transformers.

Indeed, metacomputation is *purposely* avoided in [7, 14, 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', their composition $M \circ M'$ — 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 $T_M \ \mathsf{Id} = M$ and $T_{M'} \ \mathsf{Id} = M'$, where $(T_M \circ T_{M'})\mathsf{Id}$ 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 one wants: *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.

The next section explains the advantages for modular compilation of metacomputation-based language specification over the monolithic style. Section 3 reviews the most relevant related work. In Section 4, we review the theory of monads and monad transformers and their use in language specification. Section 5 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. Section 6 shows how to combine these compiler building blocks into a compiler for the combined language, and presents a compiler and an example compilation. Section 7 discusses the impact of metacomputation-based specification on compiler correctness. Finally, Section 8 summarizes this work and outlines future research.

2 Why Metacomputations?

In this section, we will describe at a high level why two monads are better than one for modular compilation. Using metacomputations instead of a single monolithic monad simplifies the use of the "code store" (defined below) in the specification of reusable compiler building blocks.

In [8], we borrowed a technique from denotational semantics[26] for modeling jumps, namely storing command continuations in a "code store" and denoting "`jump` L" as "execute the continuation at label L in the code store." Viewing command continuations as machine code is a common technique in semantics-directed compilation[28, 25]. Because our language specifications were in monadic style, it was a simple matter to add label generator and code store states to the underlying monad. Indeed, the primary use for monads in functional programming seems to be that of adding state-like features to purely functional languages and programs[27, 22], and the fact that we structured our monads in [8] with monad transformers made adding the new states simple.

The use of a code store is integral to the modular compilation technique described in [8]. We use it to compile control-flow and procedures, and the presence of the code store in our language specifications allowed us to make substantial improvements over Reynolds[25] (e.g., avoiding infinite programs through jumps and labels). Yet the mixing of static with dynamic data into one "monolithic" monad causes a number of problems with using the code store. Consider the program "**if** b **then** (**if** b' **then** c)". Compiling the outer "**if**" with initial continuation **halt** and label 0 will result in the continuation "$[\![$**if** b' **then** $c]\!]$; **halt**" being stored at label 0 and the label counter being incremented. The problem here is that trying to compile this continuation via partial evaluation will fail. Why? Because having been *stored* rather than *executed*, the continuation stored at label 0 will not have access to the next label 1. Instead, the partial evaluator will try to increment a (dynamic) variable rather than an actual (static) integer, and this will cause an error (a partial evaluator can evaluate "1+1" but not "x+1"). In [8], the monolithic style specifications forced all static data to be explicitly passed to stored command continuations, although this was at the expense of modularity. In fact to compile **if-then-else**, the snapback operator[23] had to be used. These complications also make reasoning about compilers constructed in [8] difficult. We shall demonstrate in Section 5 that using metacomputations results in vastly simpler compiler specifications and that this naturally makes them easier to reason about.

3 Related work

Espinosa [7] and Hudak, Liang, and Jones [14] use monad transformers to create modular, extensible interpreters. Liang [13, 15] 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[21]. 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[20] applies two-level λ-calculus to code generation for a typed λ-calculus, and Nielson[21] presents an algorithm for static analysis of a typed λ-calculus which converts one-level specifications into two-level specifications. Mogensen[16] 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[21]) 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 [21, 16], 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' [25] 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 approach to compilation 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.

4 Monads and Monad Transformers

In this section, we review the theory of monads [18, 27] and monad transformers [7, 14]. 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):

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

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:

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

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 $\mathit{Expression} \to \mathsf{M}\,\mathit{int}$:

$$[\![e_1{+}e_2]\!] \;=\; [\![e_1]\!] \;\star\; (\lambda i.[\![e_2]\!] \;\star\; (\lambda j.\mathbf{unit}(i+j)))$$

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*[19] and *high-level semantics*[12].

Identity Monad Id:

$\mathsf{Id}\,\tau = \tau$

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

$x \star_{\mathsf{Id}} f = f\, x$

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

$\mathsf{M}'\tau = \mathcal{T}_{\mathsf{Env}}\, Env\, \mathsf{M}\,\tau = Env \to \mathsf{M}\tau$

$\mathbf{unit}_{\mathsf{M}'}\, x = \lambda\rho : Env.\, \mathbf{unit}_{\mathsf{M}}\, x$

$x \star_{\mathsf{M}'} f = \lambda\rho : Env.\, (x\,\rho) \star_{\mathsf{M}} (\lambda a. f\, a\, \rho)$

$\mathit{lift}_{\mathsf{M}\tau \to \mathsf{M}'\tau}\, x = \lambda\rho : Env.\, x$

$\texttt{rdEnv} : \mathsf{M}' Env$

$\texttt{rdEnv} = \lambda\rho : Env.\, \mathbf{unit}_{\mathsf{M}}(\rho)$

$\texttt{inEnv} : Env \to \mathsf{M}'\tau \to \mathsf{M}'\tau$

$\texttt{inEnv}\, \rho\, x = \lambda\_.\, (x\,\rho) : \mathsf{M}'\tau$

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

$\mathsf{M}'\tau = \mathcal{T}_{\mathsf{CPS}}\, ans\, \mathsf{M}\,\tau =$
$\quad (\tau \to \mathsf{M}\, ans) \to \mathsf{M}\, ans$

$\mathbf{unit}_{\mathsf{M}'}\, x = \lambda\kappa.\, \kappa\, x$

$x \star_{\mathsf{M}'} f = \lambda\kappa.\, x(\lambda a. f\, a\, \kappa)$

$\mathit{lift}_{\mathsf{M}\tau \to \mathsf{M}'\tau}\, x = x \star_{\mathsf{M}}$

$\texttt{callcc} : ((a \to \mathsf{M}'b) \to \mathsf{M}'a) \to \mathsf{M}'a$

$\texttt{callcc}\, f = \lambda\kappa. f(\lambda a. \lambda\_. \kappa\, a)\, \kappa$

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

$\mathsf{M}'\tau = \mathcal{T}_{\mathsf{St}}\, store\, \mathsf{M}\,\tau = store \to \mathsf{M}(\tau \times store)$

$\mathbf{unit}_{\mathsf{M}'}\, x = \lambda\sigma : store.\, \mathbf{unit}_{\mathsf{M}}\langle x, \sigma\rangle$

$x \star_{\mathsf{M}'} f = \lambda\sigma_0.\, (x\,\sigma_0) \star_{\mathsf{M}} (\lambda\langle a, \sigma_1\rangle. f\, a\, \sigma_1)$

$\mathit{lift}_{\mathsf{M}\tau \to \mathsf{M}'\tau}\, x = \lambda\sigma.\, x \star_{\mathsf{M}} \lambda y.\, \mathbf{unit}_{\mathsf{M}}\langle y, \sigma\rangle$

$\texttt{update} : (store \to store) \to \mathsf{M}'\texttt{void}$

$\texttt{update}\Delta = \lambda\sigma.\, \mathbf{unit}_{\mathsf{M}}\langle \bullet, \Delta\,\sigma\rangle$

$\texttt{getStore} : \mathsf{M}' store$

$\texttt{getStore} = \lambda\sigma.\, \mathbf{unit}_{\mathsf{M}}\langle \sigma, \sigma\rangle$

Fig. 3. The Identity Monad, and Environment, CPS, and State Monad Transformers

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 $\mathit{Expression} \to \mathit{int}$, and $[\![e_1{+}e_2]\!] = [\![e_1]\!] \star (\lambda i.[\![e_2]\!] \star (\lambda j.\mathbf{unit}(i+j)))$.

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

$$\begin{aligned}
\mathsf{M}_{St}\tau &= Sto \to \tau \times Sto \\
x \star f &= \lambda\sigma.\mathbf{let}\ (x', \sigma') = x\sigma\ \mathbf{in}\ f x'\sigma' \\
\mathbf{unit}\, v &= \lambda\sigma.(v, \sigma) \\
[\![e_1{+}e_2]\!] &= [\![e_1]\!] \star (\lambda i.[\![e_2]\!] \star (\lambda j.\mathbf{unit}(i+j)))
\end{aligned}$$

The $\star$ operation handles the bookkeeping of "threading" the store through the computation. Now, $[\![-]\!]$ has type $\mathit{Expression} \to Sto \to \mathit{int} \times Sto$. 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:

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

`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.

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

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

that would thread the two states through the computation, with separate "update" and "get" 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.

Moggi[17] developed the notion of monad transformers (which he called monad constructors) to solve this composition problem in a categorical setting, and this work was extended in [7, 14]. 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, 14].

In our compiler specifications, there are multiple states and environments added using the state and environment monad transformers. We distinguish the additional combinators associated with each of these monad transformers by appending the data type name of the new state or environment to the combinator. For example, there are *Env* (maps from variables to values) and *Addr* (free address counter) environments, so there are separate "read" and "in" combinators for both: `rdEnv` and `inEnv` for *Env*, and `rdAddr` and `inAddr` for *Addr*. Similarly, there are separate "update" and "get" combinators for the value and code states, *Sto* and *Code*. These are, respectively, `updateSto` and `getSto` for *Sto* and `updateCode` and `getCode` for *Code*.

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.

4.1 A Semantics for Metacomputation

We can formalize this notion of metacomputation using monads[7, 14, 18, 27] and use the resulting framework as a basis for staging computations. Given a monad

M, the *computations of type* a is the type $\mathsf{M}\,a$. So given two monads M and N, the *metacomputations of type* a is the type $\mathsf{M}(\mathsf{N}\,a)$, because the M-computation produces as a value an N-computation. This definition is not superfluous; as we have noted, $\mathsf{M} \circ \mathsf{N}$ is not generally a monad, so metacomputations are generally a different notion altogether from computations.

$\boxed{\text{Standard:}}$

$\mathsf{Dynam} = \mathsf{Id}$

$[\![-e]\!] : \mathsf{Dynam}(int) = [\![e]\!] \star_D \lambda i.\mathbf{unit}_D\ (-i)$

$\boxed{\text{Implementation-oriented/Monolithic:}}$

```
$\mathsf{Dynam} = \mathcal{T}_{\mathsf{Env}}\, Addr\, (\mathcal{T}_{\mathsf{St}}\, Sto\, \mathsf{Id})$
$Addr = int, Sto = Addr \to int$
$\mathtt{Thread} : int \times Addr \to \mathsf{Dynam}(int)$
$\mathtt{Thread}(i, a) =$
  $\mathtt{updateSto}[a \mapsto i] \star_D \lambda\_.\mathtt{rdloc}(a)$
$\mathtt{rdloc} : Addr \to \mathsf{Dynam}(int)$
$\mathtt{rdloc}(a) = \mathtt{getSto} \star_D \lambda\sigma.\mathbf{unit}_D(\sigma a)$
```

```
$\mathcal{M}ono[\![-e]\!] : \mathsf{Dynam}(int) =$
    $\mathcal{M}ono[\![e]\!] \star_D \lambda i.$
    $\mathtt{rdAddr} \star_D \lambda a.$
    $\mathtt{inAddr}\ (a+1)$
        $(\mathtt{Thread}(i, a) \star_D \lambda v.\mathbf{unit}_D\ (-v))$
```

$\boxed{\text{Metacomputation:}}$

$\mathsf{Dynam} = \mathcal{T}_{\mathsf{St}}\, Sto\, \mathsf{Id}$, $\mathsf{Static} = \mathcal{T}_{\mathsf{Env}}\, Addr\, \mathsf{Id}$

```
$\mathcal{C}[\![-e]\!] : \mathsf{Static}(\mathsf{Dynam}(int)) =$
        $\mathtt{rdAddr} \star_S \lambda a.$
        $\mathtt{inAddr}\ (a+1)$
                $(\mathcal{C}[\![e]\!] \star_S \lambda\phi_e : \mathsf{Dynam}(int).$
                    $\mathbf{unit}_S\ (\mathsf{Negate}(\phi_e, a))$
```

```
$\mathsf{Negate}(\phi_e, a) = \phi_e \star_D \lambda i.$
                  $\mathtt{Thread}(i, a) \star_D \lambda v.$
                         $\mathbf{unit}_D(-v)$
```

Fig. 4. Negation, 3 ways

5 A Case Study in Metacomputation-based Compiler Architecture: Modular Compilation for the While Language

In this section, we present several compiler building blocks. In section 6, 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. Of particular importance to the present work, Section 5.4 presents the reusable compiler building block for control flow, which demonstrates how metacomputation-based compiler architecture solves the difficulties with the monolithic approach we outlined in Section 2.

5.1 Integer Expressions Compiler Building Block

Consider the standard monadic-style specification of negation[7, 14, 27] 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:

$$[\![-e]\!] = [\![e]\!] \star_D \ \lambda i.\text{ “Store } i \text{ at } a \text{ and return contents of } a\text{” } \star_D \ \lambda v.\mathbf{unit}_D\ (-v)$$

Let us assume that this is written in terms of a monad Dynam with bind and unit operations $\star_D$ and $\mathbf{unit}_D$. 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 Dynam monad using monad transformers[7, 14] 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 (Dynam) is used to construct the domain containing both static and dynamic data.

In Figure 4, the definition of `Thread` uses `updateSto` : Dynam(`void`), which has been *lifted*[14] through the ($\mathcal{T}_{\mathsf{Env}}\, Addr$) monad transformer (i.e., redefined for $\mathcal{T}_{\mathsf{Env}}\, Addr\ (\mathcal{T}_{\mathsf{St}}\, Sto\ \mathsf{Id})$). Thus, we could have written $lift_{Addr}$(`updateSto`$[a \mapsto i]$) instead in the definition of `Thread` in Figure 4, but for the sake of readability, we assume throughout this paper that the combinators added by monad transformers are lifted appropriately.

$$\mathcal{C}[\![n]\!] = \mathbf{unit}_S\ (\mathbf{unit}_D(n))$$

$$\begin{array}{l}
\mathcal{C}[\![e_1 + e_2]\!] : \mathsf{Static}(\mathsf{Dynam}(int)) = \\
\quad \texttt{rdAddr} \star_S \ \lambda a. \\
\quad \texttt{inAddr}\ (a+2) \\
\quad\quad (\mathcal{C}[\![e_1]\!] \star_S \ \lambda \phi_1 : \mathsf{Dynam}(int). \\
\quad\quad (\mathcal{C}[\![e_2]\!] \star_S \ \lambda \phi_2 : \mathsf{Dynam}(int). \\
\quad\quad\quad \mathbf{unit}_S\ (\mathsf{Add}(\phi_1, \phi_2, a))
\end{array}
\qquad
\begin{array}{l}
\mathsf{Add}(\phi_1, \phi_2, a) = \\
\quad \phi_1 \star_D \ \lambda i. \\
\quad \phi_2 \star_D \ \lambda j. \\
\quad \texttt{Thread}(i, a) \star_D \ \lambda v_1. \\
\quad \texttt{Thread}(i, a+1) \star_D \ \lambda v_2. \\
\quad\quad\quad \mathbf{unit}_D(v_1 + v_2)
\end{array}$$

Fig. 5. Specification for Constants and Addition

In the “metacomputation”-style specification, we use two monads, Static, to encapsulate the static data, and Dynam to encapsulate the dynamic data. The meaning of the phrase is a metacomputation—the Static monad produces a computation of the Dynam monad. Clear separation of binding times is thus achieved.

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

5.2 Generating Code with Type-Directed Partial Evaluation

Scheme output from partial evaluator:

```
(lambda (store add negate read)
   (lambda (a0)
      (lambda (sto1)
         (cons nil
            ((store "Acc" (negate (read 0)))
               ((store 0 (negate (read 1))) ((store 1 7) sto1)))))))
```

Pretty printed version:

```
1 := 7; 0 := -[1]; Acc := -[0];
```

Fig. 6. Compiling "$- - 7$"

Code is generated via type-directed partial evaluation[4] using the method of Danvy and Vestergaard[5]. An example of code generation is presented in Figure 6. The code produced takes the form of a number of stores and reads from storage (underlined in the figure). For the sake of readability, we present a pretty-printed version of this code as well (and from now on, we show only the pretty-printed versions). To be more precise, we generate code for the expression e by partially evaluating:

$$\begin{array}{l} \lambda\texttt{store}.\lambda\texttt{plus}.\lambda\texttt{negate}.\lambda\texttt{read}. \\ \quad (\texttt{inAddr}\ 0\ \mathcal{C}[\![e]\!]) \ \star_S\ \lambda\phi_e. \\ \quad\quad \mathbf{unit}_S\ (\phi_e\ \star_D\ \lambda i.\texttt{updateSto}([Acc \mapsto i])) \end{array}$$

$(\texttt{inAddr}\ 0\ \mathcal{C}[\![e]\!])$ compiles e with the initial free store address of 0. The dynamic part of $\mathcal{C}[\![e]\!]$, ϕ_e, is then executed, producing an integer i, which is then stored in a register Acc. Before submitting the compilation semantics in Figures 4 and 5 to the partial evaluator, we must first translate the definitions of the compiler blocks and the Static and Dynam monads into Scheme, which is the input language of the type-directed partial evaluator. It should be clear that the definitions presented in this paper can be translated in a completely straightforward manner into Scheme. We replace "$\texttt{updateSto}[a \mapsto i]$" by "$\texttt{updateSto}(\texttt{store}(\texttt{a}, \texttt{i}))$" in the definition of $\texttt{Thread}(i, a)$ in Figure 4, "$(\sigma\, a)$" by "$(\texttt{read}\, a)$" (leaving the Sto argument out for readability's sake) in the definition of `rdLoc` in Figure 4, "$(-v)$" by "$(\texttt{negate}\, v)$", and "$(v_1 + v_2)$" by "$(\texttt{plus}\, v_1\, v_2)$" in Figures 4 and 5, respectively. The abstraction of the combinators `store`, `plus`, `negate`, and `read` ensures that these names will be left in residual code; in other words, their definitions are *intentionally* omitted to make the residual code look like machine language. The result of partial evaluation is as shown in Figure 6. This code generation technique is a monadic version of Danvy and Vestergaard's [4, 5].

5.3 Constant-Folding Integer Expressions Compiler Building Block

There is a code optimization technique in traditional compilers known as *constant-folding*[1], which recognizes constant expressions (e.g., "`1 + 2`") and evaluates them at compile-time, thereby eliminating some run-time computation. Constant-folding fits quite naturally and easily into the metacomputation setting as a reusable compiler building block, which is presented in Figure 7. If an expression e is constant (i.e., contains no variables), then it can be evaluated using the standard semantics for expressions $[\![e]\!]$ in the Static monad, and the value v produced thereby can be "boosted" to the dynamic phase with $\mathbf{unit}_S\ (\mathbf{unit}_D(v))$. This has the same effect as constant-folding. The standard semantics for expressions in the Static monad are:

$$\begin{aligned}
&[\![n]\!] = \mathbf{unit}_S\ (n)\\
&[\![-e]\!] = [\![e]\!]\ \star_S\ \lambda i.\mathbf{unit}_S\ (-i)\\
&[\![e_1 + e_2]\!] = [\![e_1]\!]\ \star_S\ \lambda i.[\![e_2]\!]\ \star_S\ \lambda j.\mathbf{unit}_S\ (i + j)
\end{aligned}$$

5.4 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 8. The first is a dual continuation "control-flow" semantics, found commonly in compilers[2]. If B is true, then the first continuation, $[\![c]\!]\ \star_D\ \kappa$, is executed, otherwise c is skipped and just κ is executed. A more implementation-oriented (informal) specification might be:

```
[[if b then c]] =
        $[\![b]\!]\ \star_D\ \lambda B.$
        "get two new labels $L_c, L_\kappa$" $\star_D\ \lambda\langle L_c, L_\kappa\rangle.$
        callcc ($\lambda\kappa.$
            "store $\kappa$ at $L_\kappa$, then ($[\![c]\!]\ \star_D$ ("jump to $L_\kappa$")) at $L_c$" $\star_D\ \lambda\_.$
            $B\langle$"jump to $L_c$", "jump to $L_\kappa$"$\rangle$)
```

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 8 presents an implementation-oriented specification of **if-then** in monolithic style (that is, where *Code* and *Label* are both added to Dynam). 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., updateCode$[L_\kappa \mapsto \kappa\bullet]$), but here is where the monolithic approach greatly complicates matters. Because the monolithic specification mixes static and 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

$$\mathsf{Static} = \mathcal{T}_{\mathsf{Env}}\, Addr\, \mathsf{Id} \qquad \mathsf{Dynam} = \mathcal{T}_{\mathsf{St}}\, Sto\, \mathsf{Id}$$

$$\begin{array}{lll}
\mathsf{constexp}(-) : Exp + \mathcal{V}ar \to \{\mathtt{true}, \mathtt{false}\} & & \\
\mathsf{constexp}(e) = \text{case } e \text{ of} & i & \Longrightarrow \mathtt{true} \\
 & e_1 + e2 & \Longrightarrow \mathsf{constexp}(e_1)\,\&\,\mathsf{constexp}(e_2) \\
 & -e' & \Longrightarrow \mathsf{constexp}(e') \\
 & x & \Longrightarrow \mathtt{false}
\end{array}$$

$$\mathsf{CF}[\![n]\!] = \mathbf{unit}_S\ (\mathbf{unit}_D(n))$$

$$\begin{array}{l}
\mathsf{CF}[\![-e]\!] = \text{case } \mathsf{constexp}(e) \text{ of} \\
\qquad\qquad \mathtt{true} \Longrightarrow [\![-e]\!]\ \star_S\ \lambda i.\mathbf{unit}_S\ (\mathbf{unit}_D(i)) \\
\qquad\qquad \mathtt{false} \Longrightarrow \mathtt{rdAddr}\ \star_S\ \lambda a.\mathtt{inAddr}\ (a+1) \\
\qquad\qquad\qquad\qquad (\mathsf{CF}[\![e]\!]\ \star_S\ \lambda \phi_e.\mathbf{unit}_S\ (\mathsf{Negate}(\phi_e, a))
\end{array}$$

$$\begin{array}{l}
\mathsf{CF}[\![e_1 + e_2]\!] = \\
\quad \text{case } \langle \mathsf{constexp}(e_1), \mathsf{constexp}(e_2)\rangle \text{ of} \\
\quad\ \langle \mathtt{true}, \mathtt{true}\rangle \Longrightarrow [\![e_1 + e_2]\!]\ \star_S\ \lambda i.\mathbf{unit}_S\ (\mathbf{unit}_D(i)) \\
\quad\ \langle \mathtt{true}, \mathtt{false}\rangle \Longrightarrow \\
\qquad\qquad \mathtt{rdAddr}\ \star_S\ \lambda a.\mathtt{inAddr}\ (a+1) \\
\qquad\qquad\quad ([\![e_1]\!]\ \star_S\ \lambda i.\mathsf{CF}[\![e_2]\!]\ \star_S\ \lambda \phi_2.\mathbf{unit}_S \left(\begin{array}{l} \phi_2\ \star_D\ \lambda j. \\ \mathtt{Thread}(j, a)\ \star_D\ \lambda v_2. \\ \quad \mathbf{unit}_D(i + v_2) \end{array}\right)) \\
\quad\ \langle \mathtt{false}, \mathtt{true}\rangle \Longrightarrow \\
\qquad\qquad \mathtt{rdAddr}\ \star_S\ \lambda a.\mathtt{inAddr}\ (a+1) \\
\qquad\qquad\quad (\mathsf{CF}[\![e_1]\!]\ \star_S\ \lambda \phi_2.[\![e_2]\!]\ \star_S\ \lambda j.\mathbf{unit}_S \left(\begin{array}{l} \phi_1\ \star_D\ \lambda i. \\ \mathtt{Thread}(i, a)\ \star_D\ \lambda v_1. \\ \quad \mathbf{unit}_D(v_1 + j) \end{array}\right)) \\
\quad\ \langle \mathtt{false}, \mathtt{false}\rangle \Longrightarrow \\
\qquad\qquad \mathtt{rdAddr}\ \star_S\ \lambda a.\mathtt{inAddr}\ (a+2) \\
\qquad\qquad\quad (\mathsf{CF}[\![e_1]\!]\ \star_S\ \lambda \phi_1.\mathsf{CF}[\![e_2]\!]\ \star_S\ \lambda \phi_2.\mathbf{unit}_S\ (\mathsf{Add}(\phi_1, \phi_2, a)))
\end{array}$$

Fig. 7. Constant-Folding Compilation Semantics for Integer Expressions

$\boxed{\text{Control-Flow:}}$

$\mathsf{Dynam} = \mathcal{T}_{\mathsf{CPS}}\ \mathtt{void}\ \mathsf{Id}$ $\qquad$ $[\![\textbf{if}\ b\ \textbf{then}\ c]\!] =$

$Bool = \forall\alpha.\alpha \times \alpha \rightarrow \alpha$ $\qquad$ $[\![b]\!]\ \star_D\ \lambda B : Bool.\mathtt{callcc}\ (\lambda\kappa.B\langle[\![c]\!]\ \star_D\ \kappa, \kappa\rangle)$

$\boxed{\text{Implementation-oriented/Monolithic:}}$

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

$Label = int, Code = \mathsf{Dynam}\ \mathtt{void}$

$\mathtt{jump}\ : Label \rightarrow \mathsf{Dynam}\ \mathtt{void}$

$\mathtt{jump}\ L = \mathtt{getCode}\ \star_D\ (\lambda\Pi : Code.\Pi\ L)$

$\mathtt{newlabel} : \mathsf{Dynam}(Label)$

$\mathtt{newlabel} =$

$\quad \mathtt{getLabel}\ \star_D\ \lambda l : Label.$

$\quad \mathtt{updateLabel}[L \mapsto L+1]\ \star_D\ \lambda\_.$

$\quad \mathbf{unit}_D(l)$

$$\begin{aligned}
&\mathcal{M}ono[\![\textbf{if}\ b\ \textbf{then}\ c]\!] : \mathsf{Dynam}(\mathtt{void}) = \\
&\qquad \mathcal{M}ono[\![b]\!]\ \star_D\ \lambda B : Bool. \\
&\qquad \mathtt{newlabel}\ \star_D\ \lambda L_\kappa. \\
&\qquad \mathtt{newlabel}\ \star_D\ \lambda L_c. \\
&\qquad \mathtt{callcc}\ (\lambda\kappa. \\
&\qquad\quad \mathtt{newSegment}(L_\kappa, \kappa\bullet)\ \star_D\ \lambda\_. \\
&\qquad\quad \mathtt{newSegment}(L_c, \mathcal{M}ono[\![c]\!]\ \star_D\ \lambda\_.\mathtt{jump}\ L_\kappa)\ \star_D\ \lambda\_. \\
&\qquad\qquad B\langle\mathtt{jump}\ L_c, \mathtt{jump}\ L_\kappa\rangle)
\end{aligned}$$

$\boxed{\text{Metacomputation:}}$

$\mathsf{Dynam} = \mathcal{T}_{\mathsf{CPS}}\ \mathtt{void}\ (\mathcal{T}_{\mathsf{St}}\ Code\ \mathsf{Id}), \mathsf{Static} = \mathcal{T}_{\mathsf{St}}\ Label\ \mathsf{Id}$

$\mathsf{IfThen} : \mathsf{Dynam}(Bool) \times \mathsf{Dynam}(\mathtt{void}) \times Label \times Label \rightarrow \mathsf{Dynam}(\mathtt{void})$

$$\begin{aligned}
\mathsf{IfThen}(\phi_B, \phi_c, L_c, L_\kappa) = {} & \phi_B\ \star_D\ \lambda B : Bool. \\
& \mathtt{callcc}\ (\lambda\kappa. \\
& \quad \mathtt{updateCode}[L_\kappa \mapsto \kappa\bullet]\ \star_D\ \lambda\_. \\
& \quad \mathtt{updateCode}[L_c \mapsto \phi_c\ \star_D\ \lambda\_.\mathtt{jump}\ L_\kappa]\ \star_D\ \lambda\_. \\
& \qquad B\langle\mathtt{jump}\ L_c, \mathtt{jump}\ L_\kappa\rangle)
\end{aligned}$$

$$\begin{aligned}
&\mathcal{C}[\![\textbf{if}\ b\ \textbf{then}\ c]\!] : \mathsf{Static}(\mathsf{Dynam}\ \mathtt{void}) = \\
&\qquad \mathcal{C}[\![b]\!]\ \star_S\ \lambda\phi_B. \\
&\qquad \mathcal{C}[\![c]\!]\ \star_S\ \lambda\phi_c. \\
&\qquad \mathtt{newlabel}\ \star_S\ \lambda L_c. \\
&\qquad \mathtt{newlabel}\ \star_S\ \lambda L_\kappa. \\
&\qquad\quad \mathbf{unit}_S\ (\mathsf{IfThen}(\phi_B, \phi_c, L_c, L_\kappa))
\end{aligned}$$

Fig. 8. if-then: 3 ways

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].

$$
\begin{array}{l}
\mathcal{C}[\![e_1 \leq e_2]\!] : \mathsf{Static}(\mathsf{Dynam}\,\mathit{Bool}) = \\
\quad \texttt{rdAddr} \star_S \lambda a. \\
\quad \texttt{inAddr}\ (a+2) \\
\quad\ \ \mathcal{C}[\![e_1]\!] \star_S \lambda\phi_{e_1}. \\
\quad\ \ \mathcal{C}[\![e_2]\!] \star_S \lambda\phi_{e_2}. \\
\quad\ \ \mathbf{unit}_S \boxed{\left(\begin{array}{l}
\phi_{e_1} \star_D \lambda i : \mathit{int}. \\
\phi_{e_2} \star_D \lambda j : \mathit{int}. \\
\texttt{Thread}(i, a) \star_D \lambda v_1. \\
\texttt{Thread}(j, (a+1)) \star_D \lambda v_2. \\
\quad \mathbf{unit}_D \left(\begin{array}{l} \lambda\langle\kappa_T, \kappa_F\rangle. \\ \quad ((v_1 \leq v_2) \rightarrow \kappa_T, \kappa_F) \end{array}\right)
\end{array}\right)}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{C}[\![\mathbf{while}\ b\ \mathbf{do}\ c]\!] : \mathsf{Static}(\mathsf{Dynam}\,\texttt{void}) = \\
\quad \mathcal{C}[\![b]\!] \star_S \lambda\phi_B. \\
\quad \mathcal{C}[\![c]\!] \star_S \lambda\phi_c. \\
\quad \texttt{newlabel} \star_S \lambda L_{test}. \\
\quad \texttt{newlabel} \star_S \lambda L_c. \\
\quad \texttt{newlabel} \star_S \lambda L_\kappa. \\
\quad \mathbf{unit}_S \boxed{\left(\begin{array}{l}
\texttt{callcc}\ \lambda\kappa. \\
\quad \texttt{updateCode}[L_\kappa \mapsto \kappa] \star_D \lambda\_. \\
\quad \texttt{updateCode}[L_c \mapsto \phi_c \star_D (\texttt{jump}\,L_{test})] \star_D \\
\quad \texttt{updateCode}[L_{test} \mapsto \phi_B \star_D \lambda B.((B\langle \texttt{jump}\,L_c, \texttt{jump}\,L_\kappa\rangle \bullet)] \star_D \\
\quad\ \ \texttt{jump}\,L_{test}
\end{array}\right)}
\end{array}
$$

Fig. 9. Specification for $\leq$ and **while**

The last specification in Figure 8 defines **if-then** as a metacomputation and is much simpler than the monolithic-style specification. Observe that Dynam 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*. **This is a central advantage of the metacomputation-based specification:** because of the separation of static and dynamic data into two monads, the complications outlined in Section 2 associated with storing command continuations in [8] (e.g., explicitly passing static data and use of a *snapback* operator[23]) are *completely* unnecessary.

Figure 9 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.

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

(In Figures 9, 10, and 11, we have set the dynamic parts of the computation in a box for emphasis.)

5.5 Block Structure Compiler Building Block

$$\begin{array}{ll}
\mathsf{Dynam} = \mathcal{T}_{\mathsf{St}}\, \mathit{Sto}\, \mathsf{Id} & \mathcal{C}[\![\mathbf{new}\ x\ \mathbf{in}\ c]\!] : \mathsf{Static}(\mathsf{Dynam}\ \mathtt{void}) = \\
\mathsf{Static} = \mathcal{T}_{\mathsf{Env}}\, \mathit{Env}\, (\mathcal{T}_{\mathsf{Env}}\, \mathit{Addr}\, \mathsf{Id}) & \quad \mathtt{rdAddr} \star_S \lambda a. \\
\mathtt{set} : \mathit{Addr} \to \mathsf{Dynam}(\mathtt{void}) & \quad \mathtt{inAddr}\ (a+1) \\
\mathtt{set}\ a = \lambda v.\mathtt{updateSto}(a \mapsto v) & \quad\quad \mathtt{rdEnv} \star_S \lambda \rho. \\
\mathtt{get} : \mathit{Addr} \to \mathsf{Dynam}(\mathtt{int}) & \quad\quad \mathtt{inEnv}\ (\rho[x \mapsto \mathbf{unit}_S\ \langle \mathtt{set}\ a, \mathtt{get}\ a\rangle])\ \mathcal{C}[\![c]\!] \\
\mathtt{get}\ a = \mathtt{getSto} \star_D \lambda\sigma.\mathbf{unit}_D(\sigma\, a) & \mathcal{C}[\![x]\!] = \mathtt{rdEnv} \star_S \lambda\rho.(\rho x)
\end{array}$$

Fig. 10. Compiler Building Block for Block Structure

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 10. The static part of this specification allocates a free stack location a, and the program variable x is bound to an accepter-expresser pair[24] 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 $(a + 1)$.

5.6 Imperative Features Compiler Building Block

$$\begin{array}{ll}
\mathsf{Dynam} = \mathcal{T}_{\mathsf{St}}\, \mathit{Sto}\, \mathsf{Id},\ \mathsf{Static} = \mathcal{T}_{\mathsf{Env}}\, \mathit{Env}\, \mathsf{Id} & \mathcal{C}[\![x := e]\!] : \mathsf{Static}(\mathsf{Dynam}\ \mathtt{void}) = \\
\mathcal{C}[\![c_1;c_2]\!] : \mathsf{Static}(\mathsf{Dynam}\ \mathtt{void}) = & \quad \mathtt{rdEnv} \star_S \lambda\rho. \\
\quad \mathcal{C}[\![c_1]\!] \star_S \lambda\phi_{c_1}. & \quad \mathcal{C}[\![x]\!] \star_S \lambda\langle acc, \_\rangle. \\
\quad \mathcal{C}[\![c_2]\!] \star_S \lambda\phi_{c_2}. & \quad \mathcal{C}[\![e]\!] \star_S \lambda\phi_e. \\
\quad\quad \mathbf{unit}_S\ \boxed{(\phi_{c_1} \star_D \lambda\_.\phi_{c_2})} & \quad\quad \mathbf{unit}_S\ \boxed{(\phi_e \star_D \lambda i : \mathit{int}.(acc\ i))}
\end{array}$$

Fig. 11. Compiler Building Block for Imperative Features

The simple imperative language includes assignment (**:=**) and sequencing (**;**). The compiler building block for this language appears in Figure 11. 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[24] *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*.

6 Combining Compiler Building Blocks

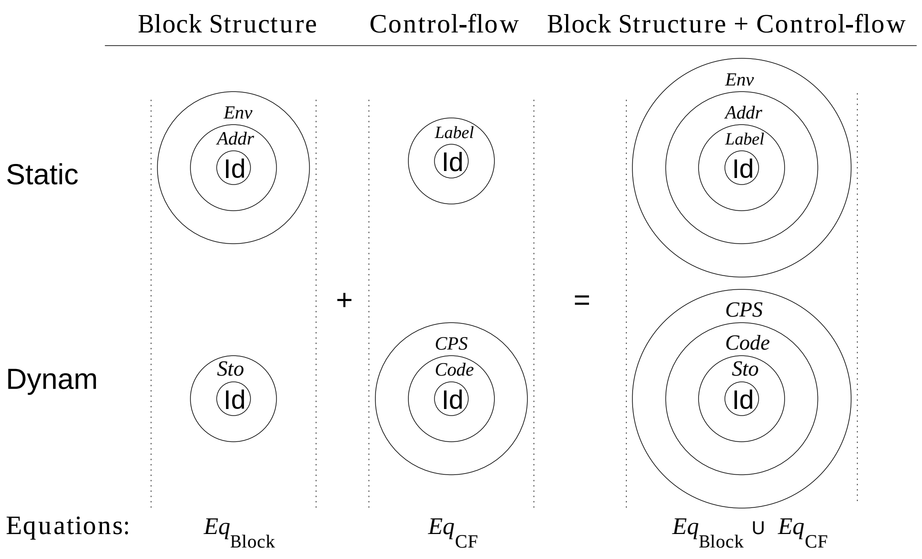

Fig. 12. Combining Compiler Building Blocks

Figure 12 illustrates the process of combining the compiler building blocks for the block structure and control-flow languages. It is important to emphasize that this is much simpler than in [8], in that there is no explicit passing of static data needed. The process is nothing more than applying the appropriate monad transformers to create the Static and Dynam monads for the combined language. Recall that for the block structure language:

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

For the control flow language:

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

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

$$\begin{aligned}\mathsf{Static} &= \mathcal{T}_{\mathsf{Env}}\, Env\, (\mathcal{T}_{\mathsf{Env}}\, Addr\, (\mathcal{T}_{\mathsf{St}}\, Label\, \mathsf{Id}))\\ \mathsf{Dynam} &= \mathcal{T}_{\mathsf{CPS}}\, \texttt{void}\, (\mathcal{T}_{\mathsf{St}}\, Code\, (\mathcal{T}_{\mathsf{St}}\, Sto\, \mathsf{Id}))\end{aligned}$$

Now, the specifications for both of the smaller languages, Eq_{Block} and Eq_{CF}, apply for the "larger" Static and Dynam monads, and so the compiler for the combined language is specified by $Eq_{Block} \cup Eq_{CF}$.

Compiler:

Dynam = $\mathcal{T}_{\mathsf{CPS}}$ `void` $(\mathcal{T}_{\mathsf{St}}\, Code\, (\mathcal{T}_{\mathsf{St}}\, Sto\, \mathsf{Id}))$, Static = $\mathcal{T}_{\mathsf{Env}}\, Env\, (\mathcal{T}_{\mathsf{Env}}\, Addr\, (\mathcal{T}_{\mathsf{St}}\, Label\, \mathsf{Id}))$
Language = Expressions + Imperative + Control-flow + Block structure + Booleans
Equations = $Eq_{\text{Expr}} \cup Eq_{\text{Imper}} \cup Eq_{\text{Control-flow}} \cup Eq_{\text{Block}} \cup Eq_{\text{Bool}}$

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;
```

Fig. 13. Compiler for While language and example compilation

Figure 13 contains the compiler for the while language, and an example program and its pretty-printed compiled version.

7 Correctness

In this section, we outline an example correctness specification for a reusable compiler building block written in metacomputation style. In particular, we illustrate the advantages with respect to compiler correctness of metacomputation-based compiler specifications over the monolithic style specifications of [8] and also of the general usefulness of monads and monad transformers with respect to compiler correctness. Although lack of space makes a full exposition of metacomputation-based compiler correctness impossible here, we hope to convey the basic issues[2].

The correctness of a reusable compiler building block for a source language feature is specified by comparing the *compilation semantics* $\mathcal{C}[\![-]\!]$ with the *standard semantics* $[\![-]\!]$ for that feature. Let us take as an example the conditional **if-then**. Its standard and compilation semantics are presented in Figure 8. A (slightly informal) specification of **if-then** is: If $L_c \neq L_\kappa$ and L_c, L_κ are unbound

[2] The interested reader may consult [9].

in the code store, then

$$\mathsf{IfThen}([\![b]\!],[\![c]\!],L_c,L_\kappa)\ \star_D\ \lambda\_.\mathtt{initCode} = [\![\mathbf{if}\ b\ \mathbf{then}\ c]\!]\ \star_D\ \lambda\_.\mathtt{initCode}$$

where $\mathtt{initCode} = \mathtt{updateCode}(\lambda\_.\Pi)$ for arbitrary constant $\Pi : \mathit{Code}$. Because $\mathsf{IfThen}([\![b]\!],[\![c]\!],L_c,L_\kappa)$ will affect the code store and $[\![\mathbf{if}\ b\ \mathbf{then}\ c]\!]$ will not, $\mathsf{IfThen}([\![b]\!],[\![c]\!],L_c,L_\kappa) \neq [\![\mathbf{if}\ b\ \mathbf{then}\ c]\!]$. But by "masking out" the code store state on both sides with `initCode`—which sets the code store to constant Π—we require that both sides of the above equation have the same action on the value store Sto.

The above specification is easier to prove than the analogous one in monolithic style because the metacomputation-based definition in Figure 8 just stores the continuation κ while the monolithic-style definition manipulates κ as was outlined in Sections 2 and 5.4. Furthermore, here is an example of how monad transformers help with compiler correctness proofs. Although the above equation holds in $\mathsf{Dynam} = \mathcal{T}_{\mathsf{CPS}}\,\mathit{Void}\,(\mathcal{T}_{\mathsf{St}}\,\mathit{Label}\,(\mathcal{T}_{\mathsf{St}}\,\mathit{Sto}\,\mathsf{Id}))$, other monad transformers could be applied to Dynam for the purposes of adding new source language features and the specification would still hold[3]. So, the use of monad transformers in this work yields a kind of proof reuse for metacomputation-based compiler correctness[15].

8 Conclusions and Future Work

Metacomputations are a simple and elegant structure for representing staged computation within the semantics of a programming language. 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 use of metacomputations in the semantics of two-level languages.

Acknowledgements

The authors would like to thank Uday Reddy and the rest of the functional programming research group at UIUC for offering many helpful suggestions that led to significant improvements in the presentation.

[3] Given certain fairly weak conditions on the order of monad transformer application. See [13–15] for details.

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 182-197.
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. W. Harrison, "Modular Compilers and Their Correctness Proofs," Doctoral Thesis (forthcoming), University of Illinois at Urbana-Champaign, 2000.
10. N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice-Hall 1993.
11. U. Jorring and W. Scherlis, "Compilers and Staging Transformations," *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1986.
12. P. Lee, *Realistic Compiler Generation,* MIT Press, 1989.
13. S. Liang, "A Modular Semantics for Compiler Generation," *Yale University Department of Computer Science Technical Report TR-1067*, February 1995.
14. S. Liang, P. Hudak, and M. Jones, Monad Transformers and Modular Interpreters. *Proceedings of the ACM Conference on the Principles of Programming Languages*, 1995.
15. S. Liang, "Modular Monadic Semantics and Compilation," Doctoral Thesis, Yale University, 1997.
16. T. Mogensen. "Separating Binding Times in Language Specifications," *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pp 12-25, 1989.
17. E. Moggi. An Abstract View of Programming Languages. *Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, 1990.*
18. E. Moggi, "Notions of Computation and Monads," *Information and Computation 93(1),* pp. 55-92, 1991.
19. P. Mosses, *Action Semantics,* Cambridge University Press, 1992.
20. 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).
21. 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.
22. S. L. Peyton-Jones and Philip Wadler. "Imperative Functional Programming," *Twentieth ACM Symposium on Principles of Programming Languages*, 1993.

23. U. Reddy. "Global State Considered Unnecessary: Semantics of Interference-free Imperative Programming," *ACM SIGPLAN Workshop on State in Programming Languages*, pp. 120-135, 1993.
24. J. Reynolds. "The Essence of Algol," *Algorithmic Languages, Proceedings of the International Symposium on Algorithmic Languages*, pp. 345-372, 1981.
25. J. Reynolds, "Using Functor Categories to Generate Intermediate Code," *Proceedings of the ACM Conference on the Principles of Programming Languages*, pages 25–36, 1995.
26. J. E. Stoy, *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory,* MIT Press, 1977.
27. P. Wadler, "The essence of functional programming," *Proceedings of the ACM Conference on the Principles of Programming Languages*, pages 1–14, 1992.
28. 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.