IEEE Computer Society International Conference on Computer Languages 1998, Loyola Univ., Chicago, May 1998

Modular Compilers Based on Monad Transformers

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

The monadic style of language specification has the advantages of modularity and extensibility: it is simple to add or change features in an interpreter to reflect modifications in the source language. It has proven difficult to extend the method to compilation. We demonstrate that by introducing machine-like stores (code and data) into the monadic semantics and then partially evaluating the resulting semantic expressions, we can achieve many of the same advantages for a compiler as for an interpreter. A number of language constructs and features are compiled: expressions, CBV and CBN evaluation of λ-expressions, dynamic scoping, and various imperative features. The treatment of recursive procedures is outlined as well. The resulting method allows compilers to be constructed in a mix-and-match fashion just as in a monad-structured interpreter.

1 Introduction and Related Work

This paper concerns the construction of modular compilers for high-level programming languages from reusable compiler building blocks. Compilers for languages with many features (e.g., expressions, procedures, etc.) are built using compiler blocks for each specific feature. Each compiler building block can be easily combined with other compiler blocks to provide compilers for non-trivial languages. Compilers constructed in this manner are modular in that source language features may be added or deleted easily, allowing the compiler writer to develop compilers at a high level of abstraction.

Modularity in our compilers comes from the fact that our language semantics is structured with *monads*[14, 19] and *monad transformers*[6, 12]. Monads may be viewed as abstract data types whose operations may be used to give denotational definitions of programming languages. Monad transformers create a new monad by extending an existing monad with additional operations, thereby allowing new programming language features to be defined while preserving any language definitions given for the original monad. It has been shown [6, 12] that modular *interpreters* may be constructed by applying the monad transformers associated with each feature in the language. Here, we extend that approach to compilers.

To obtain compilers by partial evaluation of semantic definitions[5, 8, 10], we must introduce data structures for *pass separation* [9]. This work demonstrates that, by exploiting the monadic structure of our language semantics, pass separation can be done in a modular way. Thus, language features can be combined easily, while retaining the ability to produce compiled code by partial evaluation.

In our method, we associate certain monad transformers with each language feature to be compiled. Simple expressions are compiled with the use of a stack, so expressions are associated with *two* monad transformers—one for stack address allocation and one for storing integer values. The equations defining the expression compiler block, which we refer to as the *compilation semantics*, make use of these address and value states to compile expressions. A *compiler block* for a feature is, then, the compilation semantics for the feature and its associated monad transformers. Compiler blocks may be mixed and matched just as interpreter blocks are combined in Liang, et al.[12].

To illustrate what we have accomplished, Figure 1 presents a high-level description of a modular compiler for a higher-order, imperative Algol-like language with assignment, block structure, call-by-name procedures and control flow. $\mathcal{T}_{\mathsf{Env}}$ and $\mathcal{T}_{\mathsf{St}}$ are *monad transformers*, and each transformer is labelled with the features whose compilation it is involved in. Figure 2 presents an example compilation of a program in this language. This paper provides the details for constructing the compiler blocks associated with the type declarations in Figure 1.

$$
\begin{array}{lll}
\text{Term} = & \text{Exp} & \text{— expressions} \\
& + \text{Lambda} & \text{— procedures} \\
& + \text{CF} & \text{— control flow} \\
& + \text{Block} & \text{— block structure} \\
& + \text{Imp} & \text{— simple imperative} \\
& + \text{Bool} & \text{— boolean} \\
\\
\mathbf{M_c} = & \mathcal{T}_{\mathsf{Env}}\, Env & \text{— Lambda+Block} \\
& (\mathcal{T}_{\mathsf{St}}\, Label & \text{— CF+Bool} \\
& (\mathcal{T}_{\mathsf{St}}\, Code & \text{— CF+Bool} \\
& (\mathcal{T}_{\mathsf{St}}\, Addr & \text{— Exp+Block} \\
& (\mathcal{T}_{\mathsf{St}}\, Addr & \text{— Imp+Exp} \\
& \mathbf{Id})))) &
\end{array}
$$

Figure 1: A modular compiler

Source Code:

new g:**intvar in**
let $f = \lambda v$: **intvar**. λe: **intexp**. $v := 1 + e$; $v := (g+1) + e$ **in**
$f\ g\ (g+1)$

Target Code:

```
0 := 0;                 2 := 1;
1 := 1;                 1 := [1] + [2];
2 := [0];               2 := [0];
3 := 1;                 3 := 1;
2 := [2] + [3];         2 := [2] + [3];
0 := [1] + [2];         0 := [1] + [2];
1 := [0];               halt;
```

Figure 2: Algol Example

Espinosa [6] and Hudak, Liang, and Jones [12] use monad transformers to create modular, extensible interpreters. This work shows how interpreters can be developed in a modular way, leaving open the question of whether compilers can be developed similarly. Liang [11, 13] addresses that question, proposing that monadic semantics constructed from monad transformers and monadic specifications provide a modular and extensible basis for semantics-directed compilation. He describes an experiment in [13] wherein the Glasgow Haskell compiler is retargeted to the SML/NJ back-end, and develops many examples of reasoning about monadic specifications. Since Liang does not compile to machine language, many of the issues we confront—especially pass separation—do not arise.

Jorring and Scherlis [9] introduced the term "pass separation", which introduces intermediate data structure to pass values between two phases of computation, thus enabling separation of the two phases. They showed how compilers could be constructed by introducing intermediate data structures into an interpreter and then partially evaluating. Their interpreter had no monadic structure, so that each pass separation must be done by hand.

Our work continues a long line of generating compilers from denotational semantics[8, 10, 20]. What primarily distinguishes our work is the use of monads and monad transformers to structure our semantics.

Benaissa, et al.[3], transform an interpreter for the while language into a compiler using a sequence of explicit staging annotations. The resulting compiler translates a term into a monadic intermediate code. Our work differs from theirs in that our target language is more realistic (it includes jumps, labels, etc.), we compile considerably more language features, and modularity and extensibility are key features of our approach.

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' [17] 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 Section 4, improves on Reynolds 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.

Section 2 reviews the definitions of monads and monad transformers. We illustrate the method by compiling first a simple expression language in Section 3.1, then a simple control-flow language (with only trivial expressions) in Section 3.2, and then the combination of the two in Section 3.3. The compiler for the combined language is obtained by combining the semantic equations from the smaller languages, with a small additional piece of glue. We then define call-by-value and call-by-name procedures in Section 3.4, and dynamic binding in Section 3.5, as examples of the kinds of features that can be compiled in this modular fashion. In Section 4, our final example is the idealized Algol compiled by Reynolds in [17].

2 Monads and Monad Transformers

In this section, we review the theory of monads [14, 19] and monad transformers [6, 12]. Readers familiar with these topics may skip the section, except for the last paragraph.

A *monad* is a type constructor $\mathbf{M}$ together with a

Identity Monad Id:

$$\begin{aligned}&\mathbf{Id}\,\tau = \tau\\&\mathbf{unit_{Id}}\,x = x\\&x\,\mathbf{bind_{Id}}\,f = f\,x\end{aligned}$$

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

$$\begin{aligned}&\mathbf{M}'\tau = \mathcal{T}_{\mathsf{Env}}\,Env\,\mathbf{M}\,\tau = Env \to \mathbf{M}\tau\\&\mathbf{unit_{M'}}\,x = \lambda\rho : Env.\,\mathbf{unit_M}\,x\\&x\,\mathbf{bind_{M'}}\,f = \lambda\rho : Env.\,(x\,\rho)\,\mathbf{bind_M}\,(\lambda a.f\,a\,\rho)\\&lift_{\mathbf{M}\tau\to\mathbf{M}'\tau}\,x = \lambda\rho : Env.\,x\\&\mathtt{rdEnv} : \mathbf{M}'Env = \lambda\rho : Env.\,\mathbf{unit_M}\,\rho\\&\mathtt{inEnv}(\rho : Env,\,x : \mathbf{M}'\tau) = \lambda\_.\,(x\,\rho) : \mathbf{M}'\tau\end{aligned}$$

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

$$\begin{aligned}&\mathbf{M}'\tau = \mathcal{T}_{\mathsf{St}}\,store\,\mathbf{M}\,\tau = store \to \mathbf{M}(\tau \times store)\\&\mathbf{unit_{M'}}\,x = \lambda\sigma : store.\,\mathbf{unit_M}(x,\sigma)\\&x\,\mathbf{bind_{M'}}\,f = \lambda\sigma_0 : store.\,(x\,\sigma_0)\,\mathbf{bind_M}\,(\lambda(a,\sigma_1).f\,a\,\sigma_1)\\&lift_{\mathbf{M}\tau\to\mathbf{M}'\tau}\,x = \lambda\sigma.\,x\,\mathbf{bind_M}\,\lambda y.\,\mathbf{unit_M}(y,\sigma)\\&\mathtt{update}(\Delta : store \to store) = \lambda\sigma.\,\mathbf{unit_M}(\star,\Delta\,\sigma)\\&\mathtt{rdStore} = \lambda\sigma.\,\mathbf{unit_M}(\sigma,\sigma)\end{aligned}$$

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

pair of functions (obeying certain algebraic laws that we omit here):

$$\begin{aligned}&\mathbf{bind_M} : \mathbf{M}\tau \to (\tau \to \mathbf{M}\tau') \to \mathbf{M}\tau'\\&\mathbf{unit_M} : \tau \to \mathbf{M}\tau\end{aligned}$$

A value of type $\mathbf{M}\tau$ is called a *τ-computation*, the idea being that it yields a value of type τ while also performing some other computation. The $\mathbf{bind_M}$ operation generalizes function application in that it determines how the computations associated with monadic values are combined. $\mathbf{unit_M}$ defines how a τ value can be regarded as a τ-computation; in practice, 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 $[\![-]\!]$:*Expression*$\to$ *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**, as is traditional) so that $[\![-]\!]$ has type *Expression* $\to \mathbf{M}\,int$:

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

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

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*[15] and *high-level semantics*[10].

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 *Expression*$\to$ *int.*

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

$$\begin{aligned}\mathbf{M}_{St}\tau &= Sto \to \tau \times Sto\\ x\,\mathbf{bind}\,f &= \lambda\sigma.\;\mathbf{let}\;(x',\sigma') = x\sigma\;\mathbf{in}\;fx'\sigma'\\ \mathbf{unit}\,v &= \lambda\sigma.(v,\sigma)\\ \mathtt{add}\,(x,y) &= \lambda\sigma.(x+y,\sigma)\end{aligned}$$

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

$$\begin{aligned}&\mathtt{updateSto} &&: (Sto \to Sto) \to \mathbf{M}_{St}\mathtt{void}\\& &&: f \mapsto \lambda\sigma.(\star, f\sigma)\\&\mathtt{rdSto} &&: \mathbf{M}_{St}Sto\\& &&: \lambda\sigma.(\sigma,\sigma)\end{aligned}$$

`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 $\star$). `rdSto` returns the store.

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

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

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

The key contribution of the recent work [6, 12] on *monad transformers* is to solve this composition problem. When applied to a monad $\mathbf{M}$, a monad transformer $\mathcal{T}$ creates a new monad $\mathbf{M}'$. For example, the state monad transformer, $\mathcal{T}_{St}\,store$, is

For monad $\mathbf{M}$:

$$\begin{array}{rl}
\kappa \in & [\![\mathbf{compl}]\!] = \mathbf{M}\,\mathtt{void} \\
\beta \in & [\![\mathbf{intcompl}]\!] = int \to [\![\mathbf{compl}]\!] \\
e \in & [\![\mathbf{intexp}]\!] = \mathbf{M}([\![\mathbf{intcompl}]\!] \to [\![\mathbf{compl}]\!]) \\
 & [\![\mathbf{boolcompl}]\!] = \mathbf{M}([\![\mathbf{compl}]\!] \times [\![\mathbf{compl}]\!]) \\
B \in & [\![\mathbf{boolexp}]\!] = \mathbf{M}([\![\mathbf{boolcompl}]\!] \to [\![\mathbf{compl}]\!]) \\
\phi \in & [\![\mathbf{comm}]\!] = \mathbf{M}([\![\mathbf{compl}]\!] \to [\![\mathbf{compl}]\!]) \\
 & [\![\tau_1 \to \tau_2]\!] = \mathbf{M}([\![\tau_1]\!] \to [\![\tau_2]\!])
\end{array}$$

Figure 4: Types and their Denotations

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 $\mathcal{T}_{St}\, Sto\, \mathbf{Id}$ is identical to the state monad, but, more importantly, we get a useful notion of composition: $\mathcal{T}_{St}\, Sto\, (\mathcal{T}_{St}\, Sto\, \mathbf{Id})$ is equivalent to the two-state monad $\mathbf{M_{2St}}\,\tau$.

The state monad transformer also provides `updateSto` and `rdSto` operations appropriate to the newly-created monad. Furthermore, the existing combinators of $\mathbf{M}$ are *lifted through* $\mathcal{T}_{St}$, meaning that they are redefined in a canonical manner for $\mathbf{M}'$. If L were a language whose semantics were written in terms of $\mathbf{unit}_{\mathbf{M}}$, $\mathbf{bind}_{\mathbf{M}}$, and the combinators of $\mathbf{M}$, then this semantics would remain valid for $\mathbf{M}'$. The definitions of these combinators are given in Figure 3. Other monad transformers have these same properties—they create a new monad with some auxiliary operations and a *lift* operation to redefine the operations of $\mathbf{M}$ canonically for $\mathcal{T}\mathbf{M}$;[1] the environment monad transformer is another example, also shown in Figure 3.

Finally, we note that in our examples we often make use of the iterated state monad transformer discussed above. In particular, because we need to manipulate storage locations explicitly, we find it convenient to distinguish the store proper from the "free location pointer." Thus, we use the composition of $\mathcal{T}_{St}\, Addr$ with $\mathcal{T}_{St}\, Sto$, which represents the threading of an address/store pair through a computation. This separation of the store into two components is really a "staging transformation;" it separates the *static* part of the state (the free locations) from the *dynamic* part (the stored values).

3 Introductory Examples

We introduce two semantic descriptions for each language, called the *standard* and *compilation* semantics, denoted $[\![-]\!]$ and $\mathcal{C}[\![-]\!]$. The standard semantics is a monadic version of the usual continuation semantics[2] for the language as it might appear in a standard textbook like Stoy[18]. For example in a continuation semantics for expressions with no variables, the meaning of a constant is:

$$[\![n]\!]\,\beta = \beta\, n$$

where $[\![n]\!] : (int \to Ans) \to Ans$. In monadic form, this becomes:

$$[\![n]\!] = \mathbf{unit}\ \lambda\beta.\ \beta n$$

The type $\mathbf{M}\,\mathtt{void}$ replaces the answer type Ans in this equation (so $\beta : int \to \mathbf{M}\,\mathtt{void}$.) The compilation semantics is an implementation-oriented semantics and is the result of performing pass separation on the standard semantics. We present the standard semantics for comparison only—the compiler blocks use only $\mathcal{C}[\![-]\!]$. Figure 4 gives abbreviations for types used throughout this paper. Note that they are parameterized by monad $\mathbf{M}$.

The following two subsections contain descriptions of the compilation method applied to two simple languages—the first being a simple expression language and the second a simple control flow language. We then combine the two languages in Subsection 3.3.

$$n \in Numeral, \qquad t \in Exp ::= n \mid -t \mid t_1 + t_2$$

Figure 5: Abstract syntax for Exp

$$\begin{array}{l}
[\![n]\!] = \mathbf{unit}\ \lambda\beta.\ \beta n \\
[\![-t]\!] = [\![t]\!]\ \ \mathbf{bind}\ \ \lambda e.\mathbf{unit}\ \lambda\beta.e(\lambda i.\beta(-i)) \\
[\![t_1 + t_2]\!] = [\![t_1]\!]\ \ \mathbf{bind}\ \ \lambda e_1.[\![t_2]\!]\ \ \mathbf{bind}\ \ \lambda e_2. \\
\qquad\qquad \mathbf{unit}\ \lambda\beta.(e_1\,(\lambda v_1.e_2\,(\lambda v_2.\beta\,(v_1 + v_2))))
\end{array}$$

Figure 6: Standard semantics for Exp

3.1 Simple Expression Language

The simple expression language consists of integer constants with negation and addition. Its syntax is given in Figure 5 and its standard semantics is given in Figure 6. This is a monadic version of the usual continuation-based specification of arithmetic expressions [18]. In Figure 6, $[\![-]\!]$ has type $Exp \to [\![\mathbf{intexp}]\!]$. Observe that the standard semantics uses only **bind** and **unit**, and so the standard semantics can be interpreted in *any* monad.

For our compilation examples, we use the abstract machine language from Reynolds[17]. The details are

[1] Some care must be taken in the order of application of monad transformers to ensure that lifting works. However, for the monad transformers used in this paper ($\mathcal{T}_{\mathsf{Env}}$ and $\mathcal{T}_{\mathsf{St}}$), application order is unimportant. Cf. Liang, et al.[12].

[2] There is a considerable body of work using continuations to structure compilers[2, 17, 20].

$$\mathcal{C}[\![n]\!] = \textbf{unit}\ \lambda\beta.\,\beta n$$

$$\begin{aligned}\mathcal{C}[\![-t]\!] = \mathcal{C}[\![t]\!]\ &\textbf{bind}\ \lambda e. \\ &\textbf{unit}\ \lambda\beta.e(\lambda i.\texttt{CreateTemp}(i)\ \textbf{bind}\ \lambda v. \\ &\quad \texttt{deAlloc}\ \textbf{bind}\ \lambda\_.\beta(-v))\end{aligned}$$

$$\begin{aligned}\mathcal{C}[\![t_1 + t_2]\!] = \mathcal{C}[\![t_1]\!]\ &\textbf{bind}\ \lambda e_1.\mathcal{C}[\![t_2]\!]\ \textbf{bind}\ \lambda e_2. \\ &\textbf{unit}\ \lambda\beta.(e_1\,(\lambda v_1.\texttt{CreateTemp}(v_1)\ \textbf{bind}\ \lambda v_1'. \\ &\quad e_2\,(\lambda v_2.\texttt{CreateTemp}(v_2)\ \textbf{bind}\ \lambda v_2'. \\ &\quad\quad \texttt{deAlloc}\ \textbf{bind}\ \lambda\_.\texttt{deAlloc}\ \textbf{bind}\ \lambda\_. \\ &\quad\quad\quad \beta\,(v_1' + v_2'))))\end{aligned}$$

$$\begin{aligned}&\texttt{deAlloc} = \texttt{updateA}(\lambda a.a-1) \\ &\texttt{allocLoc} = \texttt{rdAddr}\ \textbf{bind}\ \lambda a.\texttt{updateA}(\lambda a.a+1)\ \textbf{bind}\ \lambda\_.\textbf{unit}\,a \\ &\texttt{CreateTemp}(v) = \\ &\quad \texttt{allocLoc}\ \textbf{bind}\ \lambda a. && \texttt{/*allocate Addr*/} \\ &\quad \texttt{updateSto}[a \mapsto v]\ \textbf{bind}\ \lambda\_. && \texttt{/*store v at a*/} \\ &\quad \texttt{rdSto}\ \textbf{bind}\ \lambda\sigma. && \texttt{/*get curr. Sto*/} \\ &\quad\quad \textbf{unit}(\sigma\,a) && \texttt{/*return val at a*/}\end{aligned}$$

Figure 7: Compilation semantics for *Exp*

not important here; examples such as those in Figures 8 and 12 should be self-explanatory. (Square brackets should be read as "contents of.")

Given an expression like "$--1$", a compiler might generate machine code (assuming no "constant folding" optimization is performed) like:

$$tmp{:=}-1\ ;\ \phi(-[tmp])$$

for some ϕ representing the context where the expression occurs.

The behavior of the semantic definition of negation and of the compiled code are quite different; namely, the machine code stores an intermediate value in a temporary location, while the standard semantics knows nothing of locations or storage. However, it is quite simple to add address allocation and store using the state monad transformer[12, 19]. In the parlance of pass separation, the addresses and storage are *intermediate data structures.*

So, our first change to the standard semantics is to apply two state monad transformers adding addresses and value storage. Assuming that the standard semantics is written in terms of the monad $\mathbf{M}$, we can add the intermediate data structure for pass separation [9] by applying $\mathcal{T}_{\mathrm{St}}$ twice: $\mathbf{M}_c = \mathcal{T}_{\mathrm{St}}\ \mathit{Addr}\ (\mathcal{T}_{\mathrm{St}}\ \mathit{Sto}\ \mathbf{M})$ where $\mathit{Addr} = \mathit{int}$ and $\mathit{Sto} = \mathit{Addr} \rightarrow \mathit{int}$. We will call the new combinators in $\mathbf{M_c}$: `updateA`, `updateSto`, `rdAddr`, `rdSto`.

Because $\mathbf{M_c}$ has addresses and storage, we can thread each intermediate value of an expression through the store. Thus, the meaning of an expression according to the *compilation semantics* will *behave* like that of the machine code produced by the compilation of the expression, in the sense that it

Scheme output from partial evaluator:

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

Pretty printed version:

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

Figure 8: Compiling "$--1$"

stores intermediate values. We define a new combinator, $\texttt{CreateTemp}(v)$, which allocates a temporary address l, stores v in l, and returns the contents of l. The compilation semantics for the expression language is given in Figure 7[3]. Each intermediate value is first passed to `CreateTemp`, which then threads it through the store. Since `CreateTemp` just stores and retrieves a value without changing it, the correctness of the compilation semantics—that is, its equivalence to the standard semantics—is intuitively clear.

An example is presented in Figure 8. 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 t by partially evaluating:

$$\begin{aligned}&\lambda\texttt{store}.\lambda\texttt{add}.\lambda\texttt{negate}.\lambda\texttt{read}. \\ &\quad \texttt{updateA}(\lambda\_.0)\ \textbf{bind}\ \lambda\_. \\ &\quad \mathcal{C}[\![t]\!]\ \textbf{bind}\ \lambda e. \\ &\quad\quad e(\lambda i.\texttt{updateSto}([\mathit{Acc} \mapsto i])))\end{aligned}$$

The `updateA` term establishes the initial free address in the store. The initial continuation places the value of t into the accumulator. Before submitting the compilation semantics in Figure 7 to the partial evaluator, we replace "$\texttt{updateSto}[a \mapsto v]$" by "`updateSto(store(a,v))`", "$(\sigma\,a)$" by "$(\texttt{read}\,a)$" (leaving the *Sto* argument out for readability's sake), "$(-v)$" by "$(\texttt{negate}\,v)$", and "$(v_1' + v_2')$" by "$(\texttt{add}\,v_1'\,v_2')$". The abstraction of the combinators `store`, `add`, `negate`, and `read` ensure 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 8. This code generation technique is a monadic version of Danvy and Vestergaard's [4, 5].

[3] $[a \mapsto v] : \mathit{Sto} \rightarrow \mathit{Sto}$ is the usual update function which changes the value of a store at a.

$n \in Numeral$
$c \in Cmd ::= c_1;c_2 \mid Addr\texttt{:=}Exp \mid \textbf{if } Bool \textbf{ then } c$
$t \in Exp ::= n$
$b \in Bool ::= True \mid False \mid \textbf{not } b \mid b \textbf{ or } b'$

Figure 9: Abstract syntax for CF

$\boxed{[\![t : Bool]\!] : [\![\mathbf{boolexp}]\!]}$

$[\![\mathbf{True}]\!] = \mathbf{unit}\ \lambda\langle\kappa_T, \kappa_F\rangle.\ \kappa_T$

$[\![\mathbf{False}]\!] = \mathbf{unit}\ \lambda\langle\kappa_T, \kappa_F\rangle.\ \kappa_F$

$[\![\mathbf{not}\, b]\!] = [\![b]\!]\ \ \mathbf{bind}\ \ \lambda B.\mathbf{unit}\ \lambda\langle\kappa_T, \kappa_F\rangle.\ B\ \langle\kappa_F, \kappa_T\rangle$

$\boxed{[\![t : Cmd]\!] : [\![\mathbf{comm}]\!]}$

$[\![t_1;t_2]\!] = [\![t_1]\!]\ \ \mathbf{bind}\ \ \lambda\phi_1.[\![t_2]\!]\ \ \mathbf{bind}\ \ \lambda\phi_2.\mathbf{unit}(\phi_2 \circ \phi_1)$

$[\![l := t]\!] = [\![t]\!]\ \ \mathbf{bind}\ \ \lambda e.$
$\quad\quad \mathbf{unit}(\lambda\kappa.e\ (\lambda v.\texttt{updateSto}[l \mapsto v]\ \ \mathbf{bind}\ \ \lambda\_.\kappa))$

$[\![\mathbf{if}\ b\ \mathbf{then}\ c]\!] = [\![b]\!]\ \ \mathbf{bind}\ \ \lambda B.\mathbf{unit}(\lambda\kappa.B\langle[\![c]\!] \bullet \kappa, \kappa\rangle)$

$\boxed{[\![t : Exp]\!] : [\![\mathbf{intexp}]\!]}$

$[\![n]\!] = \mathbf{unit}\ \lambda\beta.\ \beta n$

$(\gamma : M\ (\tau_1 \rightarrow \tau_2)) \bullet (\psi : \tau_1) = \gamma\ \ \mathbf{bind}\ \ (\lambda\phi.\phi\,\psi)$

Figure 10: Standard semantics for CF

3.2 The Simple Control Flow Language

We now consider the compilation of a simple control-flow language with sequencing, direct assignments, and an **if-then** conditional. The syntax and standard semantics of this language are presented in Figures 9 and 10, respectively. (For simplicity, our assignment statement assigns directly to addresses rather than to variables; variables are treated, orthogonally, in Section 4.) Because we have assignment, the semantics can only be interpreted in a state monad, that is, a monad $\mathbf{M}$ that includes the combinators `updateSto` and `rdSto`.

For boolean expressions, we use a dual completion "control-flow" semantics, reflecting common practice in compilers [2].

A conditional "**if** b **then** c" would typically be compiled into machine code of the form:

<code for b>
branch on b to L_c
jump to L_{end}
L_c: <code for c>
L_{end}: κ

In moving from the standard semantics to the compilation semantics, we confront the following problem: the standard semantics for the conditional would result in

$\mathcal{C}[\![\mathbf{if}\ b\ \mathbf{then}\ c]\!] =$
$\quad \mathcal{C}[\![b]\!]\ \ \mathbf{bind}\ \ \lambda\beta.$
$\quad \texttt{newlabel}\ \ \mathbf{bind}\ \ \lambda L_{end}.\ \texttt{newlabel}\ \ \mathbf{bind}\ \ \lambda L_c.$
$\quad \mathbf{unit}(\lambda\kappa.\ \ \texttt{newSegment}(L_{end}, \kappa)\ \ \mathbf{bind}\ \ \lambda\_.$
$\quad\quad\quad \texttt{newSegment}(L_c, (\mathcal{C}[\![c]\!] \bullet \texttt{jump}\, L_{end}))\ \ \mathbf{bind}\ \ \lambda\_.$
$\quad\quad\quad\quad \beta < \texttt{jump}\, L_c, \texttt{jump}\, L_{end} >)$

$\texttt{newlabel} =$
$\quad \texttt{rdLabel}\ \ \mathbf{bind}\ \ \lambda l.\texttt{updateL}(\lambda l.l+1)\ \ \mathbf{bind}\ \ \lambda\_.\mathbf{unit}\, l$

$\texttt{jump}\, l = \texttt{rdCode}\ \ \mathbf{bind}\ \ \lambda\pi.(\pi l)$

$\texttt{getStaticContext} =$
$\quad \texttt{rdLabel}\ \ \mathbf{bind}\ \ \lambda l.\texttt{rdCode}\ \ \mathbf{bind}\ \ \lambda\pi.\mathbf{unit}\ <l, \pi>$

$\texttt{setStaticContext} <l, \pi>= \texttt{updateL}(\lambda\_.l)\ \ \mathbf{bind}\ \ \lambda\_.\texttt{updateC}(\lambda\_.\pi)$

$\texttt{snapback}\, \kappa = \texttt{rdSto}\ \ \mathbf{bind}\ \ \lambda\sigma.$
$\quad \kappa\ \ \mathbf{bind}\ \ \lambda\_.\texttt{updateSto}(\lambda\_.\sigma)$

$\texttt{newSegment}(l, \kappa) = \texttt{getStaticContext}\ \ \mathbf{bind}\ \ \lambda SC.$
$\quad \texttt{updateC}[l \mapsto (\texttt{setStaticContext}(SC)\ \ \mathbf{bind}\ \ \lambda\_.\kappa)]\ \ \mathbf{bind}\ \ \lambda\_.$
$\quad\quad \texttt{snapback}(\kappa)$

Figure 11: Compilation semantics for CF

a *tree* of instructions rather than a *sequence*, due to κ being passed to both components of $<([\![c]\!] \bullet \kappa), \kappa>$ (c.f., Figure 10), and would also result in considerable *code duplication*, since κ may be repeated many times[4]. Of course, an actual compiler would emit code once and then emit appropriate jumps. Here, a technique from denotational semantics [18] is applicable: when a command jumps to a label, it invokes the continuation stored at that label. We make this continuation store explicit, just as we made the value store explicit to compile the expression language.

As before, we add two states: one to generate new labels, and one for the continuation store proper. So, the first change to the standard semantics is to apply state monad transformers adding label and continuation stores. Given a monad $\mathbf{M}$, the monad for the compilation semantics is defined as: $\mathbf{M}_c = \mathcal{T}_{\mathrm{St}}\ Code\ (\mathcal{T}_{\mathrm{St}}\ Label\ (\mathcal{T}_{\mathrm{St}}\ Sto\ \mathbf{M}))$, where $Label = int$ and $Code = Label \rightarrow \mathbf{M}_c\,\texttt{void}$. The new combinators of $\mathbf{M}_c$ will be called `updateC` and `rdCode` (for $Code$) and `updateL` and `rdLabel` (for $Label$). We define two new combinators, `newlabel` and `newSegment`, with which we transform the definition of the conditional. `newlabel` increments the current free label, and then returns the current free label, while $\texttt{newSegment}(l, \kappa)$, roughly speaking, stores the continuation κ at l in the $Code$ state.

The compilation semantics for **if-then** and **or** are shown in Figure 11; for all other features, the compilation and standard semantics are the same. The new equation for **if-then** is very similar to the informal

[4]This is exactly what happens in Reynolds[17].

```
0: jump 1;  1: 100 := 7; jump 2;  2: halt;
```

Figure 12: Compiling: "**if** $True$ **then** 100:=7"

```
0: jump 1;           0 := [0]+[1];
1: 0 := 1;           100 := -[0];
   1 := 2;           jump 2;
   0 := [0]+[1];  2: halt;
   1 := 3;
```

Figure 13: Compiling: "**if** $True$ **then** 100:=$-((1+2)+3)$"

code outline for the compilation of **if-then** above. The only point that needs clarification is the jump to L_{end} appearing after the code for c. The idea is that the continuation store contains segments of code that are not inherently connected, so that every segment is either the completion of the entire computation or ends by invoking another segment. (Again, this is a characteristic of continuation stores as used in Stoy[18].)

The most difficult part of the compilation semantics for CF is the definitions of the auxiliary operations. Define the *static context* to be the data structures added for pass separation, in this case the label and code store. Functions `getStaticContext` and `setStaticContext` simply get and set these values. `snapback` has the rather odd job of running a command κ and then restoring the original contents of the value store; the effect is to execute κ only for its effect on the static context; we will see why this is needed momentarily.

The most subtle of these functions is `newSegment`. It is used to solve the following difficulty: When a new code segment κ is placed in the code store at label L, the subsequent computations must see the updated "next label" value so that subsequent code segments will be placed at new labels. κ itself may update this value many times as it is compiled, as it is among the values that are "threaded through" the computation. So, to get this next label, we need to execute κ. However, κ contains updates to *all* values, static and dynamic; it cannot be executed without updating both the static values and the dynamic ones (specifically, the value store). `newSegment` wants to store κ at L and execute κ for its *static* effect only. It does this by running κ in the current static context and then calling `snapback` to undo κ's effect on the dynamic store.

Code is generated as in Subsection 3.1, except that now updates and reads to and from the code store are residualized. An example compilation is presented in Figure 12.

$\boxed{\text{For lambda expression } t : \tau,\ [\![t]\!] : \mathbf{M}([\![\tau]\!])}$

$[\![v]\!] = \mathcal{C}[\![v]\!] = \texttt{rdEnv}\ \mathbf{bind}\ \lambda\rho.\rho\, v$

$[\![\lambda_n s.t]\!] = \mathcal{C}[\![\lambda_n s.t]\!] =$
$\quad \texttt{rdEnv}\ \mathbf{bind}\ \lambda\rho.\mathbf{unit}(\lambda c.\ \texttt{inEnv}(\rho[s \mapsto c], \mathcal{C}[\![t]\!]))$

$\mathcal{C}[\![t_1\, t_2]\!] = \texttt{rdEnv}\ \mathbf{bind}\ \lambda\rho.\mathcal{C}[\![t_1]\!]\ \mathbf{bind}\ \lambda f.f\ (\texttt{inEnv}(\rho, \mathcal{C}[\![t_2]\!]))$

$[\![\lambda_v s : \tau \rightarrow \tau'.\, t]\!]\ =\ \mathcal{C}[\![\lambda_v s : \tau \rightarrow \tau'.\, t]\!] =$
$\quad \texttt{rdEnv}\ \mathbf{bind}\ \lambda\rho.$
$\quad \mathbf{unit}(\lambda c.c\ \mathbf{bind}\ \lambda\phi.\texttt{inEnv}(\rho[s \mapsto \mathbf{unit}\,\phi], \mathcal{C}[\![t]\!]))$

$[\![\lambda_v s : \mathbf{intexp}.\, t]\!]\ =\ \mathcal{C}[\![\lambda_v s : \mathbf{intexp}.\, t]\!] =$
$\quad \texttt{newAddr}\ \mathbf{bind}\ \lambda a.\texttt{rdEnv}\ \mathbf{bind}\ \lambda\rho.$
$\quad \mathbf{unit}(\lambda c.c\ \mathbf{bind}\ \lambda\phi.$
$\qquad \phi(\lambda v.\texttt{updateSto}[a \mapsto v])\ \mathbf{bind}\ \lambda\_.$
$\qquad \texttt{inEnv}(\rho[s \mapsto \mathbf{unit}(\lambda\beta.\beta(\texttt{read}\,a))], \mathcal{C}[\![t]\!]))$

$[\![\mathbf{let}\, x = g\, \mathbf{in}\, f]\!] = [\![(\lambda x.f)\, g]\!]$

Figure 14: Standard and Compilation Semantics for CBN and CBV functional languages

3.3 Combining Simple Expressions with Control Flow

The additional intermediate data structure occurring in the compilation semantics for the simple expression language of Subsection 3.1 was the type *Addr* of addresses and the type *Sto* of integer storage. In the compilation of the control-flow language in Subsection 3.2, the additional structure was the type *Label* of labels and the type *Code* of continuation storage. Because of the use of monad transformers, it is a simple matter to create a monad for the compilation semantics for the language containing both simple expressions and control-flow. The syntax of this language is the conjunction of the syntax in Figures 5 and 9. The compilation monad for this combined language has the form: $\mathbf{M}_c = \mathcal{T}_{\mathrm{St}}\ Label\ (\mathcal{T}_{\mathrm{St}}\ Code\ (\mathcal{T}_{\mathrm{St}}\ Addr\ (\mathcal{T}_{\mathrm{St}}\ Sto\ \mathbf{M})))$. The semantic specification for each construct in this combined language is *identical* to its definition in Figures 7 and 11 with the proviso that the definition of `getStaticContext` and `setStaticContext` must now pass and set, respectively, the current free address as well as the current label and code store:

$\texttt{getStaticContext} =$
$\quad \texttt{rdLabel}\ \mathbf{bind}\ \lambda l.\texttt{rdCode}\ \mathbf{bind}\ \lambda\pi.\texttt{rdAddr}\ \mathbf{bind}\ \lambda a.$
$\quad \mathbf{unit}\ < l, \pi, a >$

$\texttt{setStaticContext} < l, \pi, a > =$
$\quad \texttt{updateL}(\lambda\_.l)\ \mathbf{bind}\ \lambda\_.\texttt{updateC}(\lambda\_.\pi)\ \mathbf{bind}\ \lambda\_.\texttt{updateA}(\lambda\_.a)$

(This is the "piece of glue" referred to in the introduction.) An example compilation in the combined language is presented in Figure 13.

CBN evaluation:	CBV evaluation:
`0 := 1;`	`1 := 1;`
`1 := 2;`	`2 := 2;`
`0 := [0]+[1];`	`1 := [1]+[2];`
`1 := 3;`	`2 := 3;`
`0 := [0]+[1];`	`1 := [1]+[2];`
`0 := -[0];`	`0 := -[1];`
`1 := 1`	`1 := [0];`
`2 := 2;`	`2 := [0];`
`1 := [1]+[2];`	`Acc := [1]+[2];`
`2 := 3;`	
`1 := [1]+[2];`	
`1 := -[1];`	
`Acc := [0]+[1];`	

Figure 15: Compiling: "$(\lambda i.i + i)(-((1+2)+3))$"

3.4 The Call-By-Name and Call-By-Value λ-calculus

We create a simple functional language by adding new terms for lambda expressions, application, and lambda variables to the expression language in Subsection 3.1. We include abstractions for call-by-name (CBN) and call-by-value (CBV) functions. Semantically, this requires that *environments* be added to the underlying monad, and we accomplish this by applying the environment monad transformer $\mathcal{T}_{\mathsf{Env}}$[12]. So, the monad for the compilation semantics has the form: $\mathbf{M}_c = \mathcal{T}_{\mathsf{Env}}\ Env\ (\mathcal{T}_{\mathsf{St}}\ Addr\ (\mathcal{T}_{\mathsf{St}}\ Sto\ \mathbf{M}))$. The *Addr* and *Sto* states are only necessary for CBV. For CBN procedures alone, $\mathbf{M}_c = \mathcal{T}_{\mathsf{Env}}\ Env\ \mathbf{M}$ suffices. The resulting monad $\mathbf{M}_c$ had two additional combinators: **rdEnv** : $\mathbf{M}_c Env$ and **inEnv** : $Env \times \mathbf{M}_c\tau \rightarrow \mathbf{M}_c\tau$. **rdEnv** reads the current environment, and **inEnv**$(\rho,\ x)$ evaluates x in environment ρ.

Figure 14 contains the standard and compilation semantics for the CBV and CBN λ-calculus. As in the first half of Reynolds[17], this compilation semantics corresponds to treating λ-expressions as *open* procedures, that is, procedures that are expanded like macros. Observe that the CBN standard and compilation semantics are *identical* (as is the case in Reynolds[17]). The standard and compilation semantics for CBV λ-calculus are identical at higher type, but for arguments of type **intexp**, the value of the expression argument is stored in a temporary location a. An example compilation is presented in Figure 15. Observe that with CBN evaluation, the argument $-((1+2)+3)$ is calculated *twice*, and with CBV evaluation, the argument is calculated, stored in address `0`, and *used* twice.

3.5 Dynamic Scope

In the previous language, *static* scoping was assumed. Dynamic scoping can be introduced by altering the specification for application:

$$[\![t_1\, t_2]\!] = \mathcal{C}[\![t_1\, t_2]\!] = \mathcal{C}[\![t_1]\!]\ \textbf{bind}\ \lambda f.f\ (\mathcal{C}[\![t_2]\!])$$

Observe that t_2 will not be evaluated in the current environment as with static scoping. An example is presented in Figure 16. The case of dynamic binding, when the body of f is evaluated, the most recent binding of s (i.e., 5) is stored in `1`, rather than the value of s when f was defined (i.e., 10).

p: $\ \ \textbf{let}\ s = 10\ \textbf{in}\ \textbf{let}\ f = \lambda x.x + s\ \textbf{in}$
$\quad\quad \textbf{let}\ g = \lambda s.\, f\,(s + 11)\ \textbf{in}$
$\quad\quad\quad (g\, 5)$

CBN + static scope:	CBN + dynamic scope:
`0 := 5;`	`0 := 5;`
`1 := 11;`	`1 := 11;`
`0 := [0]+[1];`	`0 := [0]+[1];`
`1 := 10;`	`1 := 5;`
`ACC := [0]+[1];`	`ACC := [0]+[1];`

Figure 16: Compiling p with static and dynamic scope

$i \in Var$
$n \in Numeral$
$c \in Cmd ::= \textbf{new}\ Var\textbf{:intvar in}\ c \mid c_1\ \textbf{;}\ c_2 \mid i \textbf{ := } e \mid \textbf{skip} \mid \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_1$
$e \in Exp ::= n \mid i \mid -e \mid e_1 + e_2$
$b \in BoolExp ::= e_1 = e_2 \mid \textbf{not}\ b$
$t \in Lambda ::= e \mid b \mid c \mid i \mid \lambda\ i.\ t \mid t_1\ t_2$

Figure 17: Abstract syntax of the source language

4 Compilation of Idealized Algol

Figure 17 contains the abstract syntax for a higher-order, imperative, CBN, Algol-like language. This is the language compiled by Reynolds in [17], and the compiler we derive is essentially identical to that in [17] for the non-recursive fragment. With the exception of the **new** operator, and the inclusion of integer variables, we have considered the compilation of each of these constructs in Section 3, and their compilation is treated *identically* for the imperative, λ-calculus, and boolean parts of the language. For arithmetic expressions, we can give, following Reynolds, a more space-efficient compilation semantics than that presented in Figure 7, but space constraints prevent our describing it here. (Please refer to Harrison, et al. [7]).

The monad for the compilation semantics for this language must contain all of the intermediate data structure included when the features were compiled in isolation in Section 3: $\mathcal{T}_{\mathsf{Env}}\ Env\ (\mathcal{T}_{\mathsf{St}}\ Label\ (\mathcal{T}_{\mathsf{St}}\ Code\ (\mathcal{T}_{\mathsf{St}}\ Addr\ (\mathcal{T}_{\mathsf{St}}\ Sto\ \mathbf{Id}))))$.

$$
\begin{aligned}
\llbracket \mathbf{new}\ i\ \mathbf{in}\ c \rrbracket = {} & \mathbf{unit}(\lambda\kappa.\texttt{rdEnv}\ \mathbf{bind}\ \lambda\rho.\texttt{allocLoc}\ \mathbf{bind}\ \lambda S. \\
& \quad \texttt{inEnv}(\rho[i \mapsto \mathbf{unit} < set\ S, get\ S >], \\
& \quad\quad \llbracket c \rrbracket\ \mathbf{bind}\ \lambda\phi.\phi(\texttt{deAlloc}\ \mathbf{bind}\ \lambda\_.\kappa)))
\end{aligned}
$$

$$
\begin{aligned}
\llbracket i := t \rrbracket = {} & \texttt{rdEnv}\ \mathbf{bind}\ \lambda\rho.(\rho\, i)\ \mathbf{bind}\ \lambda < ia, \_ > . \llbracket t \rrbracket\ \mathbf{bind}\ \lambda e. \\
& \quad \mathbf{unit}(\lambda\kappa.(ia\kappa)\ \mathbf{bind}\ \lambda\beta.(e\,\beta))
\end{aligned}
$$

$$
\begin{aligned}
& set\, l = \lambda\,\kappa.\mathbf{unit}\,(\lambda v.\texttt{updateSto}[l \mapsto v]\ \mathbf{bind}\ (\lambda\_.\,\kappa)) \\
& get\, l = \lambda\,\beta.\texttt{rdSto}\ \mathbf{bind}\ (\lambda\sigma.\,\beta\,(\sigma\, l))
\end{aligned}
$$

Figure 18: Standard Semantics for Algol: Imperative features

$$
\begin{aligned}
& \llbracket \mathbf{letrec}\ \iota \equiv c : \mathbf{comm}\ \mathbf{in}\ c' \rrbracket = \\
& \quad \texttt{rdEnv}\ \mathbf{bind}\ \lambda\rho.inEnv\ \rho[\iota \mapsto\ fix(\lambda\iota.\llbracket c \rrbracket)]\ \llbracket c' \rrbracket
\end{aligned}
$$

$$
\begin{aligned}
& \mathcal{C}\llbracket \mathbf{letrec}\ \iota \equiv c : \mathbf{comm}\ \mathbf{in}\ c' \rrbracket = \texttt{rdEnv}\ \mathbf{bind}\ \lambda\rho. \\
& \quad \texttt{currentFrame}\ \mathbf{bind}\ \lambda f.\texttt{newlabel}\ \mathbf{bind}\ \lambda L_c.\mathcal{C}\llbracket c \rrbracket\ \mathbf{bind}\ \lambda\phi. \\
& \quad \texttt{newSegment}(L_c, \texttt{inEnv}\ \rho[\iota \mapsto \texttt{mkCall}\ L_c\ f]\ \phi(\texttt{return}\,[f+1, 1])) \\
& \quad\ \mathbf{bind}\ \lambda\_.(\texttt{inEnv}\ \rho[\iota \mapsto \texttt{mkCall}\ L_c\ f]\ \mathcal{C}\llbracket c' \rrbracket)
\end{aligned}
$$

$$
\begin{aligned}
& \texttt{mkCall}\, L_c\ f = \\
& \quad \texttt{newlabel}\ \mathbf{bind}\ \lambda L_\kappa.\mathbf{unit}\lambda\kappa. \\
& \quad\quad \texttt{newSegment}(L_\kappa, \kappa)\ \mathbf{bind}\ \lambda\_. \\
& \quad\quad \texttt{updateSto}([f+1, 1] \mapsto L_\kappa)\ \mathbf{bind}\ \lambda\_.\texttt{call}\, L_c
\end{aligned}
$$

$$
\begin{aligned}
& \texttt{rdSeg}\, L = \texttt{rdCode}\ \mathbf{bind}\ \lambda\pi.\pi l \\
& \texttt{call}\, L_c = \texttt{rdSeg}\, L_c \\
& \texttt{return}\,[f, e] = \texttt{rdloc}\,[f, e]\ \mathbf{bind}\ \lambda L_\kappa.\texttt{rdSeg} L_\kappa \\
& \texttt{currentFrame} = \texttt{updateA}(\lambda a.a)\ \mathbf{bind}\ \lambda <f, d> .\mathbf{unit} f
\end{aligned}
$$

Figure 19: Standard and Compilation Semantics for recursive bindings of type **comm**

The command "**new** i **in** c" creates a new integer variable i which exists only during the execution of c. Its standard semantics is presented in Figure 18. A location S is allocated and the integer variable i is bound to an *acceptor-expresser* pair which writes and reads to and from S as in Reynolds[16]. `deAlloc` is used to deallocate S. The specification for assignment looks up the acceptor-expresser pair associated with i, and passes the appropriate continuation on to the integer expression e. The compilation semantics for **new** and assignment are identical to their standard semantics. Furthermore, the standard and compilation semantics for all other language features are as given earlier. We have shown in Figure 2 an example compilation for this language.

4.1 Recursive Bindings

In Section 3.4, procedures were compiled via inlining, but that approach will not suffice for recursive bindings. Furthermore, inlining may, in some cases, drastically increase target code size for non-recursive procedures, so some alternative method is desirable. We now consider the compilation of recursive bindings of type **comm**—command-valued procedures with no arguments. Procedures of higher type may be compiled along similar lines, but for the sake of exposition, we consider only the simplest case.

A command c may be viewed as a map ϕ : **compl** $\to$ **compl**, where the completion argument acts as a "return address". In an interpreter, ϕ may be applied to a variety of completions, but this will not suffice in a compiler since there is no direct representation for ϕ. Instead we use the following pass separation transformation which applies ϕ to a generic argument of type **compl**, resulting in a term i : **compl** which intuitively represents ϕ with "holes" in it. These holes (encapsulated as `return` in Figure 19) are computations which read the return label from a fixed location S in the current stack frame. A call to c (encapsulated as `mkCall` in Figure 19) first stores its completion κ at some label L_κ, then stores L_κ at S, and jumps to i. The type Σ of stack locations are non-negative integer pairs $<frame, disp>$, which may be thought of as a *display* address [1], or alternatively, as a *stack shape* [17]. Intuitively, $frame$ points to an activation record and $disp$ is an offset within that record. Following Reynolds[17], we assume each activation record has a *call block* where the labels of argument parameters are stored. Call block entries are denoted $[frame, e]$ which points to the *e-th* argument in the activation pointed to by $frame$. We replace the store Sto with an (unspecified) stack type $Stack$. The monad for $\mathcal{C}\llbracket - \rrbracket$ is: $\mathcal{T}_{\mathsf{Env}}\ Env\ (\mathcal{T}_{\mathsf{St}}\ Label\ (\mathcal{T}_{\mathsf{St}}\ Code\ (\mathcal{T}_{\mathsf{St}}\ \Sigma\ (\mathsf{State}\ Stack\ \mathbf{Id}))))$. Only those combinators which directly deal with addresses—namely, `CreateTemp` and `deAlloc`—must be rewritten to reflect the use of a stack. Figure 19 contains the standard and compilation semantics for **letrec** and Figure 20 presents an example.

5 Conclusions and Further Work

The main contribution of this work is the development a "mix-and-match" compilation method similar to the modular interpreter constructions of [6, 12, 19]. Using monads and monad transformers to structure semantics-directed compilers achieves much of the same flexibility and modularity that one associates with monadic interpreters. One of the advantages of the monadic approach is that the underlying denotational *model* can be made arbitrarily complex through application of monad transformers without complicating the denotational *description* unnecessarily. Because of this *separability* as Lee calls it [10], monadic specifications are a natural setting for pass separation. The use of partial evaluation in the compiler clarifies the relationship between the compilation semantics and the machine language semantics. One of the remaining issues with this compilation method is

new i **new** j **in**
i:=1 ; j:=5 ; **letrec** $\iota \equiv (i := i * j$; $j := j$-1 ; **if** $j = 0$ **then** ι) **in** ι

```
0: <0,0> := 0;                       <0,1> := [<0,2>]-[<0,3>];
   <0,1> := 0;                       <0,2> := [<0,1>];
   <0,0> := 1;                       <0,3> := 0;
   <0,1> := 5;                       brEq [<0,2>] [<0,3>] 2 3;
   storeCB [1,1] 7;               2: jump 5;
   call 1;                        3: jump 4;
1: <0,2> := [<0,0>];              4: storeCB [1,1] 6;
   <0,3> := [<0,1>];              5: return [1,1];
   <0,0> := [<0,2>]*[<0,3>];      6: jump 5;
   <0,2> := [<0,1>];              7: halt;
   <0,3> := 1;
```

Figure 20: Compiling a recursive procedure

to establish a formal correctness proof. Claims have been made [6, 11, 13] that proofs about monadic specifications retain some modularity as well, and a correctness proof of our method would be a good test of this. Another goal for future research is the compilation of language features such as exceptions and objects.

Acknowledgements

The authors would like to thank Olivier Danvy for many constructive comments about this work, and for implementing his type-directed partial evaluator. 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] Z. Benaissa, T. Sheard, and W. Taha, "Compilers as staged monadic interpreters," Oregon Graduate Institute, 1997. Submitted for publication.

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

[7] W. Harrison and S. Kamin, "Deriving Compilers from Monadic Semantics," Unpublished manuscript available at http://www-sal.cs.uiuc.edu/~harrison/pubs/proposal.ps.Z.

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

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

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

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

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

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

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

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

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

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

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

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

[20] 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.