“Bridging the Gulf: A Common Intermediate Language for ML and Haskell”
“From Interpreter to Compiler using Staging and Monads”
“Multi-State Programming: Axiomatization and Type Safety”
“The Anatomy of a Component Generator”
"Optimizing ML Using a Hierarchy of Monadic Types"
Bridging the gulf: a common intermediate language for ML and Haskell

Simon Peyton Jones
University of Glasgow and Oregon Graduate Institute

Mark Shields
University of Glasgow and Oregon Graduate Institute

John Launchbury
Oregon Graduate Institute

Andrew Tolmach
Portland State University

Abstract

Compilers for ML and Haskell use intermediate languages that incorporate deeply-embedded assumptions about order of evaluation and side effects. We propose an intermediate language into which one can compile both ML and Haskell, thereby facilitating the sharing of ideas and infrastructure, and supporting language developments that move each language in the direction of the other. Achieving this goal without compromising the ability to compile as good code as a more direct route turned out to be much more subtle than we expected. We address this challenge using monads and unpointed types, identify two alternative language designs, and explore the choices they embody.

1 Introduction

Functional programmers are typically split into two camps: the strict (or call-by-value) camp, and the lazy (or call-by-need) camp. As the discipline has matured, though, each camp has come more and more to recognise the merits of the other, and to recognise the huge areas of common interest. It is hard, these days, to find anyone who believes that laziness is never useful, or that strictness is always bad. While there are still pervasive stylistic differences between strict and lazy programming, it is now often possible to adopt lazy evaluation at particular places in a strict language (Okasaki [1996]), or strict evaluation at particular points in a lazy one (for example, Haskell's strictness annotations [Peterson et al. [1997]]).

This rapprochement has not yet, however, propagated to our implementations. The insides of an ML compiler look pervasively different to those of a Haskell compiler. Notably, sequencing and support for side effects and exceptions are usually implicit in an ML compiler's intermediate language (IL), but explicit (where they occur) in a Haskell compiler (Launchbury & Peyton Jones [1995]). On the other hand, thunk formation and forcing are implicit in a Haskell compiler's intermediate language, but explicit in an ML compiler. These pervasive differences make it impossible to share code, and hard to share results and analyses, between the two styles.

To say that "support for side effects are implicit in an ML compiler's IL" (for example) is not to say that an ML compiler will take no notice of side effects: on the contrary, an ML compiler might well perform a global analysis that identifies pure sub-expressions (though in practice few do). However, one might wonder whether the analysis would discover all the pure sub-expressions in a Haskell program translated into the IL. In the same way, if an ML program were translated into a Haskell compiler's IL, the latter might not discover all the occasions in which a function argument was guaranteed to be already evaluated. This thought motivates the following question: could we design a common compiler intermediate language (IL) that would serve equally well for both strict and lazy languages? The purpose of this paper is to explore the design space for just such a language.

We restrict our attention to higher order, polymorphically typed intermediate languages. There is considerable interest at the moment in type-directed compilation for polymorphic languages, in which type information is maintained accurately right through compilation and even on to run time (Harper & Morrisett [1995]; Shao & Appel [1995]; Tarditi et al. [1996]). Hence we focus on higher order, statically typed source languages, represented in this paper by ML (Milner & Tofte [1990]) and Haskell (Peterson et al. [1997]).

At first we expected the design to be relatively straightforward, but we discovered that it was not. In particular, making sure that the IL has good operational properties for both strict and lazy languages turns out to be rather subtle. Identifying these subtleties is the main contribution of the paper:

- We employ monads to express and delimit state, input/output, and exceptions (Section 3). Using monads in this way is now well known to theorists (Moggi [1991]) and to language designers (Launchbury & Peyton Jones [1995]; Peyton Jones & Wadler [1993]; Wadler [1992b]), but, with one exception, no compiler that we know has monads built into its intermediate language.

- We employ unpointed types to express the idea that an expression cannot diverge (Section 3.1). We show that the straightforward use of unpointed types does not lead to a good implementation (Section 3.6). This leads us to explore two distinct language designs. The first, $L_1$, is mathematically simple, but cannot be compiled well (Section 3). An alternative design, $L_2$, adds operational significance to unpointed types, by guaranteeing that a variable of unpointed type is evaluated (Section 4); this means $L_2$ can be compiled well, but weakens its theory.

- We identify an interaction between unpointed types, polymorphism, and recursion in $L_1$ (Section 3.5).
terestingly, the problem turns out to be more easily solved in $L_2$ than $L_1$ (Section 4.2).

None of these ingredients are new. Our contribution is to explore the interactions of mixing them together. We emerge with the core of a practical IL that has something to offer both the strict and lazy community in isolation, as well as offering a common framework. Our long-term goal is to establish an intermediate language that will enable the two communities to share both ideas (analyses, transformations) and systems (optimisers, code generators, run-time systems, profilers, etc.) more effectively than hitherto.

2 The ground rules

We seek an intermediate language (IL) with the following properties:

- It must be possible to translate both (core) ML and Haskell into the IL. Extensions that add laziness to ML, or strictness to Haskell, should be readily incorporated. We make no attempt to treat ML’s module system, though that would be a desirable extension.

- In order to accommodate ML and Haskell the IL’s type system must support polymorphism. This ground rule turns out to have very significant, and rather unfortunate, impact upon our language designs (Section 3.5), but it seems quite essential. Nearly all existing compilers generate polymorphic target code, and although researchers have experimented with compiling away polymorphism by type specialisation (Jones [1994]; Tolmach & Oliva [1997]), problems with separate compilation and potential code explosion remain unresolved.

- The IL should be explicitly typed (Harper & Mitchell [1992]). We have in mind a variant of System F (Girard [1990]), with its explicit type abstractions and applications. The expressiveness of System F really is required. For example, there are several reasons for wanting polymorphic arguments to functions: the translation of Haskell type classes creates “dictionaries” with polymorphic components: we would like to be able to simulate modules using records (Jones [1996]); rank-2 polymorphism is required to express encapsulated state (Launchbury & Peyton Jones [1995]); and data-structure fusion (Gill, Launchbury & Peyton Jones [1993]).

IL programs can readily be type-checked, but there is no requirement that one could infer types from a type-erased IL program.

- The IL should have a single well-defined semantics. On the face of it, compilers for both strict and lazy languages already use a common language, namely the lambda calculus. But this similarity is only at the level of syntax; the semantics of the two calculi differ considerably. In particular, the code generator from a strict-language compiler would be completely unusable in a lazy-language compiler, and vice versa. Our goal is to have a single, neutral, semantics, and hence a single optimiser and code generator.

- ML (or Haskell) programs thus compiled should be as efficient as those compiled by a good ML (resp. Haskell) compiler. In other words, compiling through the common IL should not impose any unavoidable efficiency penalty, either by way of loss of transformations (especially when starting from Haskell) or by way of a less efficient basic evaluation model (especially when starting from ML). Indeed, our hope is that we may ultimately be able to generate better code through this new route.

3 $L_1$, a totally explicit language

It is clear that the IL must be explicit about things that are implicit in “traditional” compiler ILs. Where are these implicit aspects of a “traditional” IL currently made explicit? Answer: in the denotational semantics of the IL. For example, the denotational semantics of a call-by-value lambda calculus looks something like this:

$$\varepsilon[e_1; e_2]p = (\varepsilon[e_1]p \downarrow b \downarrow) \downarrow, \quad \text{if } a = b \downarrow$$

Here, the two cases in the right-hand side deal with the possible non-termination of the argument. What is implicit in the IL – the evaluation of the argument, in this case – becomes explicit in the semantics. An obvious suggestion is therefore to make the IL reflect the denotational semantics of the source language directly, so that everything is explicit in the IL, and nothing remains to be explicated by the semantics. This is our first design, $L_1$.

Figure 1 gives the syntax and type rules for $L_1$. We note the following features:

- As a compromise in the interest of brevity all our formal material describes only a simply-typed calculus, although supporting polymorphism is one of our ground rules. The extensions to add polymorphism, complete with explicit type abstractions and applications in the term language, are fairly standard (Harper & Mitchell [1993]; Peyton Jones [1996]; Tarditi et al. [1996]). However, polymorphism adds some extra complications (Section 3.5, 3.6).

- We omit recursive data types, constructors, and case expressions for the sake of simplicity, being content with pairs and selectors.

- let is simply very convenient syntactic sugar. It is not there to introduce polymorphism, even in the polymorphic extension of the language; explicit typing removes this motivation for let.

- letrec introduces recursion. Though we only give it one binding here, our intention is that it should accommodate multiple bindings. We use it rather than a constant fix because the latter requires heavy encoding for mutual recursion that is not reflected in an implementation. We discuss recursion in detail in Section 3.5, including the unspecified side condition mentioned in the rule.

- Following Moggi [1991], we express “computational effects” — such as non-termination, assignment, exceptions, and input/output — in monadic form. The type

\footnote{We use the following standard notation. If $T$ is a complete partial order (CPO), then the CPO $T_\downarrow$, pronounced "$T$ lifted", is defined thus: $T_\downarrow = \{a_\downarrow \mid a \in T\} \cup \{\bot\}$, with the obvious ordering.}
Figure 1: Syntax and type rules for $L_1$

$M$ is the type of $M$-computations returning a value of type $\tau$, where $M$ is drawn from a fixed family of monads. The syntactic forms $let_M$ and $ret_M$ are the bind and unit combinators of the monad $M$. The only two monads we consider for now are the lifting monad, Lift, and the combination of lifting with the state transformer monad, ST. It is a straightforward extension to include the monads of exceptions and input/output as well.

This use of monads appears to contradict our goal that $L_1$ should have a trivial semantics. We discuss the reasons for this decision in Section 3.4.

Figure 2 gives the semantics of $L_1$. The semantic function $\xi$ gives the meaning of types. If it looks somewhat boring,

that is the point! The function arrow in $L_1$ is interpreted by function arrow in the underlying category of complete partial orders ($CPO$), product is interpreted by (categorical, i.e. un-lifted) product, and integers are interpreted by the integers. (If $L_1$ were expanded to have sum types, they would be interpreted by (categorical, separated) sums.) Lastly, each monad is specified by an interpretation. The monad of lifting is interpreted by lifting, while a state transformer is interpreted by a function from the current "state" to a result and the new state. The "state" is a finite mapping from location identifiers (modeled by the natural numbers, $\mathbb{N}$) to their contents.

The semantic function $\xi$ gives the meaning of expressions. Again, many of its equations are rather dull: application is interpreted by application in the underlying category, lambda abstraction by functional abstraction, and so on. The semantics of the two monads is given by their bind and unit functions. From the semantics one can prove that both $\beta$ and $\eta$ are valid with respect to the semantics, and that monadic expressions admit a number of standard transform-
motions, given in Figure 3.

### 3.1 Termination and non-termination

As we have mentioned, the interpretation of a type in $\mathcal{L}_1$ is a complete partial order (CPO). However, the interpretation of a type is not necessarily a pointed CPO; that is, the CPO does not necessarily contain a bottom element. For example, the data type of integers, $\text{Int}$, is interpreted by an unpointed CPO; that is, the interpretation of a type is not necessarily a pointed CPO. However, the interpretation of a type in $\mathcal{L}_1$ is a complete partial order (CPO). Thus, if an expression $e$ has type $\text{Int}$, then it denotes an integer, and cannot denote a non-terminating computation. How, then, do we express the type of a possibly-diverging integer-valued computation? As we have seen, $\mathcal{L}_1$ has an explicit type constructor for each monadic (i.e. computation) type, of which lifting is one. To express the type of a possibly-diverging integer we use the lifting monad. A possibly-diverging integer-valued expression therefore has type $\text{Lift \ Int}$.

So $\mathcal{L}_1$’s type system can distinguish surely-terminating expressions from possibly-diverging ones. The main reason for making this distinction in the type system is so that we can express the idea that a function takes an evaluated argument. The $\mathcal{L}_1$ lambda abstraction $\lambda x.\text{Int}.e$ expresses that $x$ cannot possibly be $\bot$, and so is a suitable translation of a lambda abstraction from a call-by-value language. On the other hand $\lambda x:\text{Lift \ Int}.e$ expresses that $x$ might perhaps be $\bot$, which fits a call-by-name or call-by-need language.

A second motivation for distinguishing pointed types from unpointed ones is that some useful program transformations that are not valid in general, hold unconditionally when one has more control over pointedness. Several researchers have explored languages that employ a distinction between pointed and unpointed types (Howard [1996]; Launchbury & Paterson [1996]), and others have explored pure languages without pointed types altogether (Cockett & Fukuhshima [1992]; Hagino [1987]; Turner [1995]). The presence of unpointed types has consequences for recursion, as we discuss in Section 3.5.

### 3.2 Stateful computations

In a similar way, we use the ST monad to express in the type system the distinction between pure and stateful computations. For example, an expression of type Lift $\text{Int}$ denotes a pure (side-effect free), albeit possibly-divergent, computation; on the other hand, and expression of type ST $\text{Int}$ denotes a computation that might diverge, or might perform some side effects on a global state and deliver an integer. Further monads can readily be added to model exceptions, or continuations, or input/output.

This use of monads is well known. Moggi pioneered the idea of using monads to encapsulate computations (Moggi [1991]; Wadler [1992a]). The lazy functional programming community has been using monads very effectively to isolate and encapsulate stateful computations and input/output within pure, lazy programs (Launchbury & Peyton Jones [1995]; Peyton Jones, Gordon & Finne [1996]; Peyton Jones & Wadler [1993]; Wadler [1992b]). Nevertheless, there are surprisingly subtle design choices to make, as we discuss in Section 3.4.

### 3.3 Translating ML and Haskell into $\mathcal{L}_1$

Before discussing its design any further, we first emphasise that $\mathcal{L}_1$’s role as a target for both strict, stateful, and pure, lazy languages by giving translations from both into $\mathcal{L}_1$. Figure 4 gives the syntax of a tiny generic source language, $\mathcal{S}$, into which we translate ML and Haskell. This gives us a prototype for either ML or Haskell, by giving it a strict or lazy interpretation respectively. In either case, $\mathcal{S}$ is assumed to have been explicitly annotated with type information by a type inference pass. The constants pair, $\text{fst}$, and $\text{snd}$ have the same (obvious) types in both interpretations. The constants $\text{new}$, $\text{read}$, and $\text{write}$ a mutable variable. Unlike pair,
their types differ in the two interpretations, as Figure 4 shows. In the lazy interpretation their types explicitly involve the source-language ST monad, and S also includes letST and retST, the unit and bind operations for ST. Modulo syntax, this is precisely how Haskell expresses stateful computation (Launchbury & Peyton Jones [1995]).

Then Figure 5 gives two translations of $S$ into $L_1$:

![Table of translations]

The "ML" translation, $\mathcal{M}$, gives the source language a stateful, strict, semantics. The result of a term translated by $\mathcal{M}$ is a computation in the ST monad, and functions also return computations in ST. That is, if the ML type system considers that $\Gamma \vdash e : \tau$, then $\mathcal{M}[e] : ST \mathcal{M}[\tau]$. The rule for application uses letST to evaluate both the function and its argument, and to sequence any state changes they contain, before applying the function to the argument. In expressions produced by the $\mathcal{M}$ translation, each variable is bound to a non-monadic type; that is, any effects (state or non-termination) are performed before binding the variable. When a variable, lambda, or pair is translated we simply return the value using retST. Lastly, a recursive ML declaration can only bind a function; hence the rule for letrec.

The "Haskell" translation, $\mathcal{H}$, gives the source language (minus the state-changing operations) a pure, non-strict semantics. A key difference from the ML translation is that the Haskell translation of data types, such as integers, pairs, and lists, are lifted, because Haskell allows values of these types to be recursively defined. Unlike the ML translation, the translation of Haskell's function type does not need to have an explicit Lift on the codomain. Nor does the translation necessarily return a Lift computation: if the Haskell type system concludes that $\Gamma \vdash e : \tau$ then $\mathcal{H}[e] : \mathcal{H}[\tau]$. $\mathcal{H}$ translates Haskell's ST-monad computations directly into $L_1$'s ST monad, just as you would hope. The only rorsome point is that the first argument of wr has source-language type Ref $\tau$, and hence has $L_1$ type Lift (Ref $\mathcal{H}[\tau]$). It must therefore be lifted into the ST monad using liftToST so that it can be evaluated in the ST monad.

It is interesting to compare the two type translations. $\mathcal{M}$ uses exactly the call-by-value translation of Wadler [1992a], with the computational effect at the end of the function arrow. On the other hand $\mathcal{H}$ does not use Wadler's call-by-name translation, as one might otherwise expect. Indeed, there is no monadic effect in the translation of function types at all; instead the Lift monad shows up in the translation of data types.

This translation of Haskell function types assumes that \lambda x:bot \rightarrow bot, where bot has value \bot, denote the same value in Haskell. Recent changes to Haskell are likely to allow these values to be distinguished, forcing a lifting of function types, and hence a more gruesome encoding of function application.

3.4 Why not encode the monads?

We have said that $L_1$ is meant to make everything explicit, so that there is nothing to be said when giving its semantics. In apparent contradiction, we made the semantics of the monads implicit — that is, explained only by the semantics of $L_1$. Why, for example, did we not make the ST monad

4The translation given here introduces quite a few "administrative redexes": a slightly more complex translation can avoid them (Sabry & Wadler [1996]).

5We do not treat the runST encapsulator of Launchbury & Peyton Jones [1995] here, but it is easy to do so.
explicit by representing a value of type \( ST \tau \) as a state-transforming function in \( \mathcal{L}_1 \), and representing \( \text{let}_{ST} \) and \( \text{ret}_{ST} \) using the other \( \mathcal{L}_1 \) forms? For example, instead of the \( \mathcal{L}_1 \) term

\[
\text{let}_{ST} x \leftarrow e \quad \text{in} \quad b
\]

we could write the \( \mathcal{L}_1 \) term

\[
\text{bind}_{ST} e \quad (x \cdot b)
\]

where \( \text{bind}_{ST} \) is defined (directly in \( \mathcal{L}_1 \)) as follows

\[
\text{bind}_{ST} = \lambda m \ k \ s . \text{let} \ p = m \ s \ \text{in} \ k \ (\text{fst} \ p) \ \text{and} \ p
\]

Here, the state passing is made explicit, but the state itself is still abstract, supporting the new, read, and write operations. This is the approach advocated by Launchbury & Peyton Jones [1995, Section 9]. It has the notable advantage that we can simplify \( \mathcal{L}_1 \) by getting rid of \( \text{let}_{M} \) and \( \text{ret}_{M} \) entirely.

We do not adopt that approach here, for three reasons:

- Encoding the monad in purely functional terms is a reasonable way of giving its semantics, but it may not be a reasonable way of giving its implementation. Consider, for example, the monad of exceptions in a strict language. The functional encoding would perform a conditional test whenever a possibly-exceptional value was bound: but the expected implementation is stack-based with no tests. Instead, a whole chunk of stack is popped when an exception is raised. Keeping the monad explicit in \( \mathcal{L}_1 \) allows the code generator to generate efficient code.

- Even where an efficient code-generation strategy does exist, its correctness may be fragile. For example, Launchbury & Peyton Jones [1995] describes an update-in-place implementation of the primitive operations (read and write) in the state monad. However, that implementation is only correct if the state is single-threaded. That is certainly the case in the terms produced by \( \mathcal{M} \), but it might not remain the case after performing \( \mathcal{L}_1 \) transformations. For example, a \( \beta \)-expansion might duplicate the state. It may be possible to preserve the single-threadedness of the state by limiting the transformations performed on the \( \mathcal{L}_1 \) program. (For example, we believe that using only transformations that are correct in a call by need calculus is sufficient. (Sabry [1997])) Even where this is true, it creates a complicated proof obligation.

- There may be useful transformations available that are specific to a particular monad (for example, swapping the order of non-interfering assignments), but which become inaccessible, or hard to spot, when expressed in a purely-functional encoding of the monad.

We find these reasons compelling. On the other hand, we were concerned that by not translating the monadic code into a core of \( \mathcal{L}_1 \) we might lose valuable transformations. So far, however, we have found no transformation that cannot be expressed in the monadic version of \( \mathcal{L}_1 \), providing the standard monad laws are implemented (Figure 3).

### 3.5 Recursion in \( \mathcal{L}_1 \)

One consequence of our decision to allow a type to be modeled by an unpointed CPO is that we have to take care with recursion. The rule (REC) in Figure 1 suggests that a \( \text{letrec} \) can be constructed at any type. But that is not so. Consider

\[
\text{letrec} \ x : \text{Int} = \ldots x \ldots \text{in} \ldots
\]

Such a recursive definition is plainly nonsense, because \( \text{Int} \) is an unpointed type and has no bottom element, so there might be no solution, or many solutions, to the recursive definition. We can only do recursion over pointed CPOs\(^6\).

How, then, can we make sense of recursion? One solution is to link recursion to the Lift monad, since Lift adds a bottom to its argument domain:

\[
\begin{align*}
\Gamma, x : \text{Lift} \tau & \vdash e_1 : \text{Lift} \tau \\
\Gamma, x : \tau & \vdash e_2 : \rho \\
\Gamma & \vdash \text{letrec} \ x : \tau = e_1 \quad \text{in} \ e_2 : \rho
\end{align*}
\]

Figure 6 gives rules for determining when a type is pointed. Unfortunately, the extension to a polymorphic type system is problematic: is the type \( \alpha \) pointed or not? There are three possible choices:

- We could decide that type variables can only range over pointed types. This is precisely the restriction proposed by Peyton Jones & Launchbury [1991], but it is unacceptable in our IL because we expect (the translations of) most ML data types to be unpointed. For example, an ordinary, non-recursive polymorphic function such as the identity function could not be applied to both \( \text{Int} \) and \( \text{ret}_{\text{Int}} \), because one has a lifted type and one does not.

\[\text{Figure 6: Rules for pointed types}\]

\[
\begin{align*}
\Gamma & \vdash (\text{Lift} \tau) \text{ pointed} \\
\Gamma & \vdash (\text{ST} \tau) \text{ pointed} \\
\Gamma & \vdash (\tau_1 \Rightarrow \tau_2) \text{ pointed} \\
\Gamma & \vdash \tau_1 \text{ pointed} \\
\Gamma & \vdash \tau_2 \text{ pointed} \\
\Gamma & \vdash \tau \text{ pointed}
\end{align*}
\]

\[\text{Figure 6 gives rules for determining when a type is pointed.}\]

\[\text{Unfortunately, the extension to a polymorphic type system is problematic: is the type } \alpha \text{ pointed or not? There are three possible choices:}\]

\[\begin{align*}
\Gamma, x : \tau & \vdash e_1 : \tau \\
\Gamma, x : \tau & \vdash e_2 : \rho \\
\Gamma & \vdash \tau \text{ pointed}
\end{align*}\]

\[\text{(RECb)}\]

\[\text{Figure 6 gives rules for determining when a type is pointed.}\]

\[\text{Unfortunately, the extension to a polymorphic type system is problematic: is the type } \alpha \text{ pointed or not? There are three possible choices:}\]

\[\begin{align*}
\Gamma, x : \tau & \vdash e_1 : \tau \\
\Gamma, x : \tau & \vdash e_2 : \rho \\
\Gamma & \vdash \tau \text{ pointed}
\end{align*}\]

\[\text{(RECb)}\]

\[\text{Figure 6 gives rules for determining when a type is pointed.}\]

\[\text{Unfortunately, the extension to a polymorphic type system is problematic: is the type } \alpha \text{ pointed or not? There are three possible choices:}\]

\[\begin{align*}
\Gamma, x : \tau & \vdash e_1 : \tau \\
\Gamma, x : \tau & \vdash e_2 : \rho \\
\Gamma & \vdash \tau \text{ pointed}
\end{align*}\]

\[\text{(RECb)}\]

\[\text{Figure 6 gives rules for determining when a type is pointed.}\]
Since the first two choices are untenable, we conclude that unpointed types, requires the use of qualified universal quantification, where type variables at which fixpoints are taken are explicitly qualified:

\[ \text{nth : } \forall \alpha . \text{Int} \rightarrow (\text{List } \alpha) \rightarrow \alpha \]

Alternatively, we could employ qualified universal quantification, where type variables at which fixpoints are taken are explicitly qualified:

\[ \text{nth : } \forall \alpha \in \text{Pointed .Int} \rightarrow (\text{List } \alpha) \rightarrow \alpha \]

Launchbury & Paterson [1996] elaborate on this idea.

Since the first two choices are untenable, we conclude that adding polymorphism to a language with both recursion and unpointed types, requires the use of qualified universal quantification.

3.6 Controlling evaluation in $\mathcal{L}_1$

While $\mathcal{L}_1$ seems to be quite suitable from a theoretical point of view, it suffers from a serious practical drawback: $\mathcal{L}_1$ is vague about the timing and degree of evaluation. Consider the $\mathcal{L}_1$ expression:

\[ \text{let } x := e \text{ in } f x \]

What code should the code generator produce for such an expression?

- An ML compiler writer would probably expect the code to evaluate the right-hand side of the let, and then call $f$ passing the value thus computed. But this eager strategy is incorrect in general if $e$ diverges, and $f$ does not evaluate its argument, as a quick glance at Figure 2 will confirm.

- A safe strategy is to build a thunk (suspension) for the right-hand side, bind $x$ to this thunk, and call $f$ passing the thunk to it. That is precisely what the code generator for a lazy language would do.

Now suppose that we are compiling code for $f$, and that $f$ has type Int $\rightarrow$ Int. The major motivation for distinguishing Int from Lift Int was to allow the compiler to treat values of type Int as certainly-evaluated, just as a strict-language compiler would assume (Section 3.1). It is unacceptable for $f$ to test whether its argument is evaluated; such a choice would guarantee that no ML compiler would use this intermediate language! Also, the safe strategy for preparing the $f$'s argument does indeed pass an unevaluated thunk, so $f$ must be prepared for this eventuality.

Can we instead use a hybrid strategy?

- A hybrid strategy for compiling let expressions might use the type of the bound variable to decide what to do: for types whose values are sure to converge (such as Int) it can evaluate the right-hand side eagerly, otherwise it can build a thunk. This strategy works for a simply-typed language but fails (again!) when we introduce polymorphism. What is the code generator to do with a let that binds a value of type $\alpha$? Either the instantiating type must be passed as an argument, or we must have two versions of the code, one for terminating types and one for possibly-diverging ones.

We regard these complications as a very serious (and far from obvious) objection to using $\mathcal{L}_1$ for operational purposes.

3.7 Summary

We expected it to be a routine matter to translate both Haskell and ML into a common language built directly on top of the standard mathematics for programming-language semantics. To our surprise it was not, as Sections 3.5-3.6 describe. $\mathcal{L}_1$ may still be quite useful as a kernel language for reasoning about programs. However, as Section 3.6 has shown, it is unsuitable as a compiler intermediate language. Thus motivated, we now turn our attention to a second design that is more suitable as an IL.

4 $\mathcal{L}_2$, a language of partial functions

Our second design starts from the problem we described in Section 3.6. Operationally, it is essential to be able to control exactly when evaluation takes place, so that the recipient of a value knows for sure whether or not it is evaluated.

Since we want to control what evaluation is done when, the obvious thing to do is to make let (and, of course, function application) eager. That is, to evaluate let $x := e$ in $b$ one evaluates $e$, binds it to $x$, and then evaluates $b$. (We use the operational term "eager", rather than the semantic term "strict" because the latter does not mean anything if the type of $e$ has no bottom element.) How, then, are we to translate the lets and function applications of a lazy language? There is a standard way to do so, namely by making the construction and forcing of thunks explicit (Friedman & Wise [1976]). This is what we do in $\mathcal{L}_2$.

Figure 2 gives the syntax and extra type rules for $\mathcal{L}_2$. There is now only one monad, ST; the Lift monad is now implicit in the semantics of $\mathcal{L}_2$ so that let and function application can be eager. There is a new syntactic form, $<e>$, that suspends the evaluation of $e$, and a new constant, force, that forces the suspension returned by its argument. There is one new type, $\langle \rho \rangle$, which is the type of $<e>$ if $e$ has type $\rho$.

The two new type rules.

Another new feature is that types are divided into value types, $\tau$, and computation types, $\rho$. Intuitively, an expression has a computation type, while a variable is always bound to a value type. Another way to say this is that the typing judgement now has the form

\[ \{ z_1 : \tau_1, \ldots, z_n : \tau_n \} \vdash e : \rho \]

The type rules of Figure 1 apply unchanged, because we carefully used $\tau$ and $\rho$ in the right places, although they were synonymous in $\mathcal{L}_1$. Function arguments and the right-hand sides of let(rec) expressions all have value types, and are evaluated eagerly. This separation of value types from computation types neatly finesse the awkward question of what it means to "evaluate" an argument computation without also "performing" it, which caused us some heart-searching in earlier un-stratified versions of $\mathcal{L}_2$. For example, the expression $(< e \text{ (read } r))$ is ill-typed, and hence we do not have to evaluate (read $x$) without also performing its state changes. Indeed, expressions of type ST $\tau$ can only occur as
Figure 7: Extra syntax and type rules for $\mathcal{L}_2$

Figure 8 gives the semantics of $\mathcal{L}_2$ in full. The crucial point is that $\mathcal{L}_2$'s function type arrow is now interpreted as the CPO of partial functions, denoted \(-\triangleright\), and the semantic evaluation function $\mathcal{E}$ takes an expression to a partial function from environments to values. Many of the equations are defined conditionally. For example, the equation for $\mathcal{E}[e_1; e_2]p$ says that if both $\mathcal{E}[e_1]p$ and $\mathcal{E}[e_2]p$ are defined, then the result is just the application of those two values; otherwise, there is no equation that applies for $\mathcal{E}[e_1; e_2]p$, so it too is undefined.

The $<\cdot>$ type constructor is modeled using lifting; the semantics of force and $<\cdot>$ move to and fro between lifted CPOs and partial functions. It may seem odd that we use two different notations — Lift $\tau$ in $\mathcal{L}_1$ and $<\tau>$ in $\mathcal{L}_2$ — with the same underlying semantic model, namely lifting. The reason is that in $\mathcal{L}_1$, we use lifting as a monad (with a bind operation, for example), whereas in $\mathcal{L}_2$ we use it to model thunks (with a force operation but no bind).

The entire semantics of $\mathcal{L}_2$ could instead be presented in the CPO of total functions, using the isomorphism:

\[ S \rightarrow T \cong S \rightarrow T_\bot \]

Which to choose is just a matter of taste. What we like about our presentation is that each $\mathcal{L}_2$ type constructor corresponds directly to a single categorical type constructor, whereas in the alternative presentation the $\mathcal{L}_2$ function type gets a more "encoded" translation. Launchbury & Baraki [1996] use partial functions in essentially the same way.

The translation of "ML" into $\mathcal{L}_2$ is exactly the same as the translation of $\mathcal{L}_1$. The translation of "Haskell" is different, however, because we now have to be explicit about the introduction of thunks (Figure 9). Concerning types, notice the use of the type constructor $<\cdot>$ on the arguments of functions and data constructors. Concerning terms, the thunk-former $<\cdot>$ is used for function arguments and the right-hand side of all let and letrec definitions. Thunks are evaluated explicitly, using force, when returning a variable or the result of the let or snd.

4.1 Controlling evaluation in $\mathcal{L}_2$

The main benefit of using $\mathcal{L}_2$ is that its semantics permit an easier interpretation of vanilla let; namely, "evaluate the right-hand side, bind the value to the variable, and then evaluate the body". A consequence is that any variable of type other than $<\cdot>$ or a type variable (which might be instantiated to $<\cdot>$), is sure to be fully evaluated, just as in any ML implementation.

4.2 Recursion in $\mathcal{L}_2$

Another advantage of $\mathcal{L}_2$ is that we can solve our earlier difficulties with recursion (Section 3.5) without requiring bounded quantification.

Firstly, we may or may not have to restrict letrecs to bind only syntactic values, because we cannot eagerly evaluate the right-hand side. (Why not? Because we cannot construct the environment in which to evaluate it.) That in turn means that the meaning of the right-hand side is always defined, which is why there is no side condition in the semantics of letrec.

But Figure 7 further restricts the right-hand side of a letrec to be a particular sort of syntactic value, a pointed value, or...
\[
\begin{aligned}
T : \text{Type} & \rightarrow \text{CPO} \\
T[\text{Int}] & = \exists \\
T[\text{Int} \rightarrow \text{Int}] & = T[\text{Int}] \rightarrow T[\text{Int}] \\
T[\text{Ref}] & = T[\text{Int}] \times T[\text{Int}] \\
T[\text{ST}] & = \text{State} \rightarrow (T[\text{Int}] \times \text{State}) \\
T[\text{Ref}] & = \mathcal{N}
\end{aligned}
\]

\[
\begin{aligned}
E : \text{Term} & \rightarrow \text{Env} \\
E[e_1 \cdot e_2] & = \rho(s) \\
E[e_1] & = k \\
E[e_1 \cdot e_2] & = (E[e_1] \cdot E[e_2]) \\
E[e_1 \cdot e_2] & = (\lambda y. E[e_1][z := y]) \\
E[e_1 \cdot e_2] & = E[e_1][z := E[e_2]] \\
E[\text{letrec } x : r = p v \text{ in } e_2] & = E[e_1](fiz(\lambda \rho, \rho[x := E[p[v]]))) \\
E[\text{letrec } x : r = p v \text{ in } e_2] & = \text{bindm} (E[e_1]) (\lambda y. E[e_2][x := y]) \\
E[\text{letrec } x : r = p v \text{ in } e_2] & = \text{unitm} (E[e_1]) \\
E[e_1] & = \bot
\end{aligned}
\]

\[
\begin{aligned}
n \text{fst} (a, b) & = a \\
n \text{snd} (a, b) & = b \\
\text{forces} a & = a \\
\text{binds} m k s & = k(r, s') \\
\text{unit} m s & = (m, s) \\
\text{new } v s & = (r, \delta(r \mapsto u)) \\
\text{rd } r s & = (s(r), s) \\
\text{wr } r s & = ((r, s), s)
\end{aligned}
\]

\[
\begin{aligned}
\text{returns } m s & = (\text{fst}(r, s), \text{fst}(r, s)) \\
\text{where } r & \notin \text{dom}(s) \\
\text{if } r & \notin \text{dom}(s) \\
\text{otherwise}
\end{aligned}
\]

Figure 8: Semantics of \(L_2\)

\(PValue\). The syntactic category of \(PValues\) is chosen so that it can only denote a value from a pointed domain, and hence a letrec definition always has a least fixpoint. To see this, consider the forms that a \(PValue\) can take:

- A lambda abstraction denotes a partial function, and the CPO of partial functions is always pointed: its least element is the everywhere undefined function.
- A thunk \(\langle e \rangle\), where \(e : \tau\), is drawn from the pointed CPO \(T[\tau]_\bot\).

Fortunately, the syntactic restriction of letrec does not lose any useful expressiveness. ML insists that letrec bind only functions (which are \(PValues\)), while Haskell binds thunks (which are also \(PValues\)). So there is no difficulty with translating the recursion arising in both ML and Haskell into \(L_1\).

### 4.3 Why not have just one monad?

Now that we have eliminated the Lift monad, and made vanilla let eager, there is another question we should ask: why not give vanilla let the semantics of \(\text{let}_{\text{ST}}\), and eliminate the latter altogether? To put it another way, we have made eager evaluation implicit in the semantics of let; why not add implicit side effects as well? After all, the code generated for \(\text{let}_{\text{ST}} x = e \text{ in } b\) will be something like "the code for \(e\) followed by the code for \(b\)", and that is just the same as the code we now expect to generate for \(\text{let } x = e \text{ in } b\).

However, if we have just one form of let we lose valuable optimising transformations. In particular, the sequence of computations in \(\text{ST}\) must be maintained, whereas let bindings can be reordered freely. Changing the order of evaluation is fundamental to several useful transformations, including common sub-expression, loop invariant computations, all kinds of code motion (Peyton Jones, Partain & Santos [1996]), inlining, and strictness analysis (remember we may be compiling a lazy language into \(L_1\)). To take a simple example, the following transformation is not in general valid for \(\text{let}_{\text{ST}}\), but is valid for vanilla let (assuming there are no name clashes):

\[
\begin{aligned}
\text{let } x_1 = e_1 \text{ in let } x_2 = e_2 \text{ in } b \\
&= \text{let } x_2 = e_2 \text{ in let } x_1 = e_1 \text{ in } b
\end{aligned}
\]

Of course, one could do an effects analysis to determine which sub-expressions were pure, as good ML compilers do, but that is effectively just what the monadic type system records!

### 5 Assessment

#### 5.1 \(L_1\) vs \(L_2\)

What have we lost in the transition from \(L_1\) to \(L_2\), apart from a somewhat more complicated semantics? One loss is \(L_1\)'s ability to describe types whose values are sure to terminate. If a \(L_1\) function has type \(\text{Int} \rightarrow \text{Int}\) then a call to the function cannot diverge; but the same is not true of \(L_2\). This does not have much impact on a compiler, but it makes programmer reasoning about \(L_2\) programs more complicated.
Another important difference is that $L_2$ has a weaker rule. $L_1$ has full $\beta$-conversion. That is, for any expressions $c$ and $b$:

$$\text{let } z = c \text{ in } b = b[c/z]$$

(A similar rule holds for application, of course.) In $L_2$, however, $\beta$ does not hold in general. A particular case of this is that if $z$ is not mentioned in $b$ then in $L_1$, the binding can be discarded; in $L_2$ the binding can only be discarded if the right-hand side is a value.

However $\beta_v$ — a restricted version form of $\beta$ that allows only values to be substituted — is valid in $L_2$. Values are defined in Figure 7, and include variables, constants, and lambda abstractions, as usual. However, values also include thunks. Hence any Haskell $\beta$ reduction has a corresponding $\beta_v$ reduction in its $L_2$ translation. Thus, the restriction to $\beta_v$ will not prevent a Haskell compiler from doing anything it can do in an implicitly lazy language with a full $\beta$ rule.

Thus far we have assumed a call-by-name semantics, in which we are content to duplicate arbitrary amounts of work provided we do not change the overall result. In practice no compiler would be so liberal; we desire a call-by-need semantics in which work is not duplicated. As Ariola et al. [1996] describes, we can give a call-by-need semantics to $L_1$ by weakening $\beta$ to $\beta_v$ and adding a garbage-collection rule that allows an unused let binding to be discarded. An analogous result holds in $L_2$: we can obtain call-by-need semantics by replacing $<e>$ by $<e>$ in the definition of values in Figure 7.

5.2 $L_2$ vs Haskell and ML ILs

Our main theme is the search for an IL that can serve for both ML- and Haskell-like languages. However, we believe that a language like $L_2$ is attractive in its own right to either community in isolation, because one might get better code from an $L_2$-based compiler.

For the Haskell compiler writer $L_2$ offers the ability to express in its type that a value is certainly evaluated. This gives a nice way to express the results of strictness analysis: a function argument of unpointed type must be passed by value. Flat arrays and strict data structures also become expressible.

For the ML compiler writer $L_2$ offers the ability to express the fact that a computation is free from side effects, which is a precondition for a raft of useful transformations (Section 4.3). While this information can be gleaned from an effects analysis, maintaining this information for every subexpression, across substantial program transformations is not easy. In $L_2$, however, local transformations can perform, and record the results of, a simple incremental effects analysis. For example, consider the following ML function:

$$\text{fun } f \ x = \text{fst} \ (\text{fst} \ x)$$

If we translate this into $L_2$ we obtain:

$$f = \text{ret} \ (\lambda x. \text{let} \ a2 = \text{let} \ s1 = \text{ret} \ x \text{ in} \ \text{ret} \text{ (fst} \ a1))$$

Simple application of the rules of Figure 3 allows this expression to simplify to:

$$f = \text{ret} \ (\lambda x. \text{let} \ a1 = x \text{ in} \ \text{let} \ a2 = \text{fst} \ a1 \text{ in} \ \text{ret} \text{ (fst} \ a2))$$

Now the $\text{ret}$ can be floated outwards, to give:

$$f = \text{ret} \ (\lambda x. \text{ret} \text{ (fst} \ a1))$$

In this form, the inner $\text{ret}$ makes it apparent that $f$ has no side effects. We have, in effect, performed a sort of incremental effects analysis. The same idea can be taken further. If $f$ is inlined at its call sites, then the $\text{ret}$ may cancel with $\text{let}$ there, and so on. Even if $f$'s body is big, we can use the "worker-wrapper" technique of Peyton Jones & Launchbury [1991] to split $f$ into a small, inlinable wrapper and a large, non-inlinable worker, $f_w$, thus:

$$f = \text{ret} \ (\lambda x. \text{ret} \text{ (fst} \ a1))$$

Blume & Appel [1997] describe a similar technique that they call "lambda-splitting".

The point of all this is that there is a real payoff for an ML compiler from making the $ST$ monad explicit. Easy, incremental transformations perform a local effects analysis; at each stage the state of the analysis is recorded in the program itself, rather than in some ad hoc auxiliary data structures; and all other program transformations will automatically preserve (or exploit) the analysis.

5.3 Parametricity

Polymorphic functions have certain parametricity properties that may be derived purely from their types (Mitchell & Meyer [1985]; Reynolds [1985]; Wadler [1989]). For example, in the pure polymorphic lambda calculus, a function $f$ with type $\forall a. a \rightarrow a$ satisfies the theorem:

$$\forall A,B. \forall h : A \rightarrow B. \forall x,y : A. h (f \ x) = f (h \ y)$$

In fact, $f$ satisfies something even stronger in which the function $h$ can be an arbitrary relation between $A$ and $B$.

When we add "polymorphic" constants to the pure calculus, the effect is that the choice of functions $h$ becomes restricted. For example, adding a fix point operator $\text{fix} : \forall a. (a \rightarrow a) \rightarrow a$ forces the restriction that the $h$ functions be strict (map $\bot$ to $\bot$) and inductive (i.e. continuous). This is the situation in Haskell, for example.

Adding polymorphic sequencing, say through an operator $\text{seq} : \forall a, \beta, \alpha \rightarrow \beta \rightarrow \beta$ or by building it into the semantics of function application, forces the restriction that the $h$ functions be bottom-reflecting (i.e. defined on all defined arguments). This is the basic situation in pure ML.

Adding polymorphic equality forces the $h$ functions to be at least one-to-one; and adding polymorphic state operations like $!r$ seems to remove any last shreds of interesting parametricity.

What, then, are the parametricity properties of $L_1$ and $L_2$? If parametricity properties are weakened by claiming various primitives to be more polymorphic than they really are, then by being more cautious in the types we assign them, we may hope to restrengthen parametricity.
In $L_2$, for example, recursion is only done either at a function type, or at a suspension type. Recursion is never permitted as a fully polymorphic type (unlike in Haskell). This has the effect of allowing the strictness side condition to be dropped, though inductiveness (or continuity) is still required. The same is achieved in $L_1$ through the use of the pointed restriction (see Launchbury & Paterson [1996] for a comparable situation). Furthermore, since all state operations are explicitly typed within the state monad, they also do not interfere with parametricity in a negative way.

The main difference between $L_1$ and $L_2$ is to do with forcing evaluation. $L_1$ has no polymorphic forcing operation, so has no consequent weakening of its parametricity property. $L_2$ does, however — it is built into its eager function application. Thus for $L_2$ the parametricity theorem demands the $h$ functions to be everywhere defined.

To see an example of this, consider the function $K: \forall \alpha, \beta. \alpha \rightarrow \beta \rightarrow \alpha$ which selects its first argument, discarding its second. The parametricity theorem is

$$\forall \alpha, \beta, \alpha' \rightarrow \beta' \rightarrow \alpha \rightarrow \beta 
\forall h_1: \alpha \rightarrow \alpha', h_2: \beta \rightarrow \beta' 
\forall x: \alpha, y: \beta 
(h_1 (K x y)) = (h_2 (K h_1 x y))$$

Clearly this holds only if $h_2$ is total (defined everywhere), otherwise the right hand side may not be defined when the left hand side is.

There is a practical implication to this. A class of techniques for removing intermediate lists called foldr-build relies on parametricity for its correctness (Gill, Launchbury & Peyton Jones [1993]). While a strictness side condition is not damaging, a totality condition is too restrictive. The technique can no longer rely on the types to provide sufficient guidance for correctness. This is disappointing, although unsurprising. The compiler can still recover the short-cut deforestation technique by refining $L_2$'s type system to use qualified types along the lines of Launchbury & Paterson [1996].

5.4 Side effects and polymorphism

It is well known that the ability to create polymorphic references can lead to unsoundness in the type system (Tofte [1990]). For example, if we are able to create a reference $x$ with type $\forall \alpha. \text{Ref } \alpha$ then we would be able to write the following erroneous code:

```
let $ref = () <- xv (r Int) 2 in
let $f: (Int->Int) <- rd (r (Int->Int)) in
$ref (f 3)
```

However in both $L_1$ and $L_2$ any expression of type $\forall \alpha. \text{Ref } \alpha$ is undefined in any environment! The only way to construct a value of $\text{Ref } \alpha$ type is with $\text{new}$, which returns a value of type $\text{ST} (\text{Ref } \tau)$. The only way to strip off the $\text{ST}$ constructor is with $\text{letS}$. Looking at the typing rule for $\text{letS}$, we can see that bound variable must have type $\text{Ref } \tau$.

SML's so-called "value restriction" conservatively restricts generalisation in $\text{let}$ bindings precisely to avoid the construction of such polymorphic references. We conjecture (though we have not proved) that $L_1$ and $L_2$ are both sound without any such side conditions.

5.5 ML thunks

One of the advantages of a language that supports both strict and lazy evaluation is that it can accommodate source languages that have such a mixture. Indeed, it is quite straightforward to map Haskell's strictness annotations (Peterson et al. [1997]) onto $L_2$. Coming from the other direction, it has long been known that thunks can be encoded explicitly in a strict, imperative language. For the sake of concreteness we use the notation proposed for ML in Okasaki [1996]. In this proposal delayed ML expressions are prefixed by a "$\text{let}$", thus:

```
let val $x = $(f y) in b end
```

Here, assuming $(f y)$ has type $\text{int}$, $x$ is bound to a thunk of type $\text{int}$ susp that, when forced, evaluates $(f y)$ and overwrites the thunk with its value.

We expected that these "ML thunks" would map directly onto $L_1$'s thunks, but that turned out not to be the case. The semantics of ML thunks is considerably more complicated than that of $L_1$'s thunks, because of the interaction with state. Consider the following ML expression:

```
let val rec $x = \{let val $y = !x - 1 in
x := $y;
if $y=0$ then 0
else force $x +$ force $x$
end
in ... end
```

(This defines $x$ recursively, which is not possible in ML, but essentially the same thing can be done using another reference to "tie the knot". We use the recursive form to reduce clutter.) When $x$ is evaluated it decrements the contents of the reference cell $r$; but then, if the new value is non-zero, $x$ evaluates itself! In effect, there can be multiple simultaneous activations of $x$, rather like the multiple activations of a recursive function. (Indeed, a non-memoising implementation of ML thunks can be obtained by representing $x$ by $A(x,e)$.) Furthermore, these multiple activations can each have a different value, because they each read the state.

$L_2$'s thunks have a much simpler semantics. A thunk has only one value, and there can be at most one activation of the thunk\(^1\). The key insight is that evaluation of a $L_2$ thunk has no side effects, unlike the ML thunk above. But what if the contents of the thunk performs side effects? For example:

```
let $x = \{let $v = Int <- rd $r in \text{wr} (v+1)\} in $e
```

Here, if $r : \text{Ref Int}$, then $x$ has type $\text{ST} (\text{Int})$, not $\text{Int}$. Forcing the thunk (with $\text{force}$) causes no side effects (apart from updating the thunk itself), and yields a computation that, when subsequently performed (by a $\text{letS}$), will increment the location $r$. The computation $x$ may be performed many times; for example, $e$ might be

```
let $al: () <- force $x in \text{letS}$ $a2: () <- force $x in ...
```

What this means, though, is that the more complicated semantics of ML thunks have to be expressed explicitly in $L_2$, presumably by coding them up using explicit references.

\(^1\)More precisely, if there is more than one then the thunk's value depends on its own value, so its value is undefined. This property justifies the well-known technique of "black-holing" a thunk, both to avoid space leaks and to report certain non-termination (Jones [1992]).
6 Related work

The FLINT language has rather similar objectives to the work described here, in that it aims to serve as a common infrastructure for a variety of higher-order typed source languages (Shao [1997b]). However, FLINT has not (so far) concentrated much on the issue of strictness and laziness, which is the main focus of this paper. The ideas described here could readily be incorporated in FLINT.

Both the Glasgow Haskell Compiler and the TIL ML compiler use a polymorphic strongly-typed internal language, though the latter is considerably more sophisticated and complex (Peyton Jones [1996]; Tarditi et al. [1996]). Neither, however, seriously attempt to compile the others main evaluation-order paradigm.

7 Further work

In this paper we have concentrated on a core calculus. Some work remains to extend it to a practical IL:

- Recursive data types and case expressions must be added — we anticipate no difficulty here.
- A proof of type soundness is needed. As we note in Section 5.4 its soundness is not obvious.
- We have a simple operational semantics for $L_1$; we are confident that it is sound and adequate, but have yet to do the proofs.
- We are studying whether is is possible to combine $L_1$'s ability to describe certainly-terminating computations with $L_2$'s operational model.

Accommodating the ML module system is likely to involve a significant extension of the type system (Harper & Stone [1997]); we have not yet studied such extensions.

In a separate paper we discuss how to use the framework of Pure Type Systems to allow the language of terms, types, and kinds to be merged into a single language and compiler data type (Peyton Jones & Meijer [1997]). We hope to merge the results of that paper and this one into a single IL.

We have made no attempt to address the tricky problem of how to combine monads. For example, ML includes the monad of state and exceptions. Is it advantageous to separate them into the composition of two monads, or is it better to have a single, combined monad? In the former case, what transformations hold?

An important operational question is that of the representation of values, especially numbers. Quite a few papers have discussed how to use unboxed representations for data values, and it would be interesting to translate their work into the framework of $L_2$ (Leroy [1992]; Peyton Jones & Launchbury [1991]; Shao [1997a]).

Acknowledgements

We would like to thank the POPL referees, Nick Benton, Bob Harper, Andrew Kennedy, Jeff Lewis, Erik Meijer, Chris Okasaki, Ross Paterson, Amr Sabry, and Tim Sheard for helpful feedback on earlier discussion and drafts of this paper. We gratefully acknowledge the support of the Oregon Graduate Institute, funded by a contract with US Air Force Material Command (F19628-93-C-0069).

References


J Launchbury & R Paterson [1996], “Parametricity and unboxing with unpointed types,” in European Symposium on Programming (ESOP’96), Linköping, Sweden, Springer-Verlag LNCS 1058, Jan 1996.


SL Peyton Jones [1996], “Compilation by transformation: a report from the trenches,” in European Symposium on Programming (ESOP’96), Linköping, Sweden, Springer-Verlag LNCS 1058, Jan 1996, 18–44.


From Interpreter to Compiler using Staging and Monads

Tim Sheard and Zine-el-abidine Benaissa
Pacific Software Research Center
Oregon Graduate Institute
P.O. Box 91000 Portland, Oregon 97291-1000 USA

April 14, 1998

Abstract

In writing this paper we had two goals. First, to promote METAML, a programming language for writing staged programs, and second, to demonstrate that staging a program can have significant benefits. We do this by example: the derivation of an executable compiler for a small language. We derive the compiler in a rigorous fashion from a semantic description of the language. This is done by staging a denotational semantics, expressed as a monadic interpreter. The compiler is a program generator, taking a program in the source language (a while-program) as input and producing an ML program as target. The ML program produced is in a restricted subset of ML over which the programmer has complete control. It is encapsulated in a special data-structure called code. The meta-programming capabilities of METAML allow this data-structure to be directly executed (run-time code generation), or to be analysed. We illustrate this analysis of generated code to build a source to source transformation which applies the monad laws to significantly improve the generated code.

1 Compilers as staged interpreters

Interpreters, when implemented in high-level declarative languages, are very close to the interpreted language's denotational semantics. Because of this, interpreters are usually used for the development of prototypes, but such prototypes lack both efficiency and any connection to the underlying system in which the compiled code must run. If expressed in a monadic style, an interpreter can be mapped closer to the underlying system, and the structuring properties of the monad even allow the interpreter to be reused as the system evolves [32, 14, 27]. Nevertheless, the effort used to build the interpreter is often considered wasteful since the programmer still needs to re-implement the compiler from scratch after building the interpreter.

Our solution to this problem is the following multi-step method. First, construct the denotational semantics as an interpreter in a functional language. Second, capture the effects of the language, and the environment in which the target language must run, in a monad. Then rewrite the interpreter in a monadic style. Third, stage the interpreter using meta-programming techniques. This staging is similar to the staging of interpreters using a partial evaluator, but is explicit rather than implicit, since the programmer places the annotations directly, rather than using an automatic binding time analysis to discover where they should be placed. This leaves programmers in complete control, and they can limit what appears in the residual program. Fourth, the resulting program is both a data-structure and a program, so it can be both directly executed and analysed. This analysis can include both source to source transformations, or translation into another form (i.e.
intermediate code or assembly language). Because the programmer has complete control over the structure of the residual program this can be a trivial task.

Staging of interpreters using partial evaluation has been done before [2, 4]. The contribution of this paper is to show that this can all be done in a single program. A system incorporating staging as a first class feature of a language is a powerful tool. While using such a tool to write a compiler the source language can be given semantics, it can be staged, translated, and optimized all in a single paradigm. It requires neither additional processes nor tools, and is under the complete control of the programmer; all the while maintaining a direct link between the semantics of interpreter and those of the compiler. Staging organizes the task of constructing a compiler into simple, incremental steps, where the semantic connection is maintained through each stage of the derivation. Each step is a relatively easy task compared to building a compiler from scratch. Constructing a compiler using a staged language has the following benefits:

- **Simplicity.** Each task is a simple one, and builds incrementally on the previous tasks.

- **Correctness.** The compiler remains connected to its semantics. Each artifact produced by a task, is provably correct with respect to the artifacts of the previous tasks. The final artifact is both a compiler for the language and a semantics equivalent to the original semantics.

- **Reuse.** Each artifact reuses the code of the previous artifact.

- **Control.** The programmer has complete control over the resulting output. He develops his program with staging in mind, and the completely controls the structure of the residual program.

2 **Staging in METAML**

METAML is almost a conservative extension of Standard ML. Its extensions include four staging annotations. To delay an expression until the next stage one places it between meta-brackets. Thus the expression `<23>` (pronounced “bracket 23”) has type `<int>` (pronounced “code of int”). We illustrate the important features of the staging annotations in the short METAML session below.

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

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

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

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

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

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

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

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

Running a piece of code, strips away the enclosing brackets, and evaluates the expression inside.

3 Monads in METAML

We assume the reader has a working knowledge of monads[30, 33]. We use the unit and bind formulation of monads[32]. In METAML a monad is a data structure encapsulating a type constructor M and the unit and bind functions.

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

This definition uses SML’s postfix notation for type application, and two non-standard extensions to ML. First, it declares that the argument ('M : * -> *) of the type constructor Monad is itself a unary type constructor [8]. We say that 'M has kind: * -> *. Second, it declares that the arguments to the constructor Mon must be polymorphic functions [21]. The type variables in brackets, e.g. ['a,'b], are universally quantified. Because of the explicit type annotations in the datatype definitions the effect of these extensions on the Hindley-Milner type inference system is well known and poses no problems for the METAML type inference engine.

In METAML, Monad is a first-class, although pre-defined or built-in type. In particular, there are two syntactic forms which are aware of the Monad datatype: Do and Return. Do and Return are METAML’s syntactic interface to the unit and bind of a monad. We have modeled them after the do-notation of Haskell[10, 24]. An important difference is that METAML’s Do and Return are both parameterized by an expression of type 'M Monad. Users may freely construct their own monads, though they should be very careful that their instantiation meets the monad axioms. Do and Return are syntactic sugar for the following:

(* Syntactic Sugar Derived Form *)

Do (Mon(unit,bind)) { x <- e; f } = bind e (fn x => f)

Return (Mon(unit,bind)) e = unit e

3
In addition the syntactic sugar of the Do allows a sequence of \( x_i <- e_i \) forms, and defines this as a nested sequence of Do's. For example:

\[
\begin{align*}
\text{Do } m \{ x_1 <- e_1; x_2 <- e_2; x_3 <- e_3; e_4 \} &= \\
\text{Do } m \{ x_1 <- e_1; \text{Do } m \{ x_2 <- e_2; \text{Do } m \{ x_3 <- e_3; e_4 \} \} \}
\end{align*}
\]

The monad laws, expressed in METAML’s Do and Return notation are:

\[
\begin{align*}
\text{Do } \{ x <- \text{Return } e; z \} &= z[e/x] \\
\text{Do } \{ x <- m; \text{Return } x \} &= m \\
\text{Do } \{ x <- \text{Do } \{ y <- a; b \}; c \} &= \text{Do } \{ y' <- a; \text{Do } \{ x <- b[y'/y]; c \} \} \\
&= \text{Do } \{ y' <- a; x <- b[y'/y]; c \}
\end{align*}
\]

4 The three-step method for compiler development

In this section, we illustrate our method by building the front end of a compiler for a small imperative while-language. We proceed in three steps. First, we introduce the language and its denotational semantics by giving a monadic interpreter as a one stage METAML program. Second, we stage this interpreter by using a two stage METAML program in order to produce a compiler. Third, we illustrate the usefulness of the staging approach, by defining a function that takes the output code of the compiler as input and returns an optimized version. This function is simply a pattern-matching based implementation of the monadic identity and associativity laws. This makes a dramatic difference in the quality of the generated code, and is completely reusable because the laws hold for any monad, not just the monad used in the example.

This illustrates the usefulness of combining the monadic and staged approaches. Without the monadic structure of the interpreter, the usefulness of the monadic-laws would have to be re-captured in a domain specific manner for every compiler. Without the structure provided by the staging, the pattern-matching based rewrite system would be impossible to use, because the compile-time computations would intervene and make recognition of the patterns impossible. In the staged interpreter, the compile-time code has disappeared by the time we want to apply the pattern based monadic-law transformer.

4.1 The while-language

In this section, we introduce a simple while-language composed from the syntactic elements: expressions (Exp) and commands (Com). In this simple language expressions are composed of integer constants, variables, and operators. A simple algebraic datatype to describe the abstract syntax of expressions is given in METAML below:

```plaintext
datatype Exp =
  Constant of int  (* 5 *)
| Variable of string (* x *)
| Minus of (Exp * Exp) (* x - 5 *)
| Greater of (Exp * Exp) (* x > 1 *)
| Times of (Exp * Exp) ; (* x * 4 *)
```

Commands include assignment, sequencing of commands, a conditional (if command), while loops, a print command, and a declaration which introduces new statically scoped variables. A declaration introduces a variable, provides an expression that defines its initial value, and limits its scope to the enclosing command. A simple algebraic datatype to describe the abstract syntax of commands is:
A simple while-program in concrete syntax, such as

declare x = 150 in
declare y = 200 in { while x > 0 do { x := x - 1; y := y - 1 }; print y}

is encoded abstractly in these datatypes as follows:

val S1 =
Declare("x",Constant 150,
  Declare("y",Constant 200,
    Seq(While(Greater(Variable "x",Constant 0),
      Seq(Assign("x",Minus(Variable "x",Constant 1)),
        Assign("y",Minus(Variable "y",Constant 1)))))
     Print(Variable "y"))));

4.2 The structure of the solution

Staging is an important technique for developing efficient programs, but it requires some
forethought. To get the best results one should design algorithms with their staged solu-
tions in mind.

The meaning of a while-program depends only on the meaning of its component ex-
pressions and commands. In the case of expressions, this meaning is a function from
environments to integers. The environment is a mapping between names (which are in-
troduced by Declare) and their values.

There are several ways that this mapping might be implemented. Since we intend to
stage the interpreter, we break this mapping into two components. The first component, a
list of names, will be completely known at compile-time. The second component, a list
of integer values that behaves like a stack, will only be known at the run-time of the compiled
program.

The functions that access this environment distribute their computation into two
stages. First, determining at what location a name appears in the name list. and second.
by accessing the correct integer from the stack at this location. In a more complicated
compiler the mapping from names to locations would depend on more than just the decla-
rarion nesting depth, but the principle remains the same. Since every variable's location
can be completely computed at compile-time, it is important that we do so, and that these
locations appear as constants in the next stage.

Splitting the environment into two components is a standard technique (often called a
binding time improvement) used by the partial evaluation community[9]. We capture this
precisely by the following purely functional implementation.

type location = int;
type index = string list;
type stack = int list;

(* position : string -> index -> location *)
fun position name index =
  let fun pos n (nm::nms) = if name = nm then n else pos (n+1) nms

in pos 1 index end;

(* fetch : location -> stack -> int *)
fun fetch n (v::vs) = if n = 1 then v else fetch (n-1) vs;

(* put: location -> int -> stack -> stack *)
fun put n x (v::vs) = if n = 1 then x::vs else v::(put (n-1) x vs);

The meaning of Com is a stack transformer and an output accumulator. It transforms
one stack (with values of variables in scope) into another stack (with presumably different
values for the same variables) while accumulating the output printed by the program.

To produce a monadic interpreter we could define a monad which encapsulates the
index, the stack, and the output accumulation. Because we intend to stage the interpreter
we do not encapsulate the index in the monad. We want the monad to encapsulate only
the dynamic part of the environment (the stack of values where each value is accessed by
its position in the stack, and the output accumulation).

The monad we use is a combination of monad of state and the monad of output.

datatype 'a M = StOut of (int list -> ('a * int list * string));
fun unStOut (StOut f) = f;
fun unit x = StOut(fn n => (x,n,""));
fun bind e f = StOut(fn n => let val (a.ni.sl) = (unStOut e) n
val (b,n2,s2) = unStOut(f a) ni
in (b,n2,s1 * s2) end);
val mswo : M Monad = Mon(unit.bind); (* Monad of state with output *)

The non-standard morphisms must describe how the stack is extended (or shrunk)
when new variables come into (or out of) scope; how the value of a particular variable is
read or updated; and how the printed text is accumulated. Each can be thought of as an
action on the stack of mutable variables, or an action on the print stream.

(* read : location -> int M *)
fun read i = StOut(fn ns => (fetch i ns.ns,""));

(* write : location -> int -> unit M *)
fun write i v = StOut(fn ns =>( (), put i v ns, "" ));

(* push: int -> unit M *)
fun push x = StOut(fn ns => ( (), x :: ns, "" ));

(* pop : unit M *)
val pop = StOut(fn (n::ns) => ((), ns, "" ));

(* output: int -> unit M *)
fun output n = StOut(fn ns => ( (), ns, (toString n)"" ));

4.3 Step 1: monadic interpreter

Because expressions do not alter the stack, or produce any output, we could give an evaluation function for expressions which is not monadic, or which uses a simpler monad than the monad defined above. We choose to use the monad of state with output throughout our implementation for two reasons. One, for simplicity of presentation, and two because if the while language semantics should evolve, using the same monad everywhere makes it easy to reuse the monadic evaluation function with few changes.

The only non-standard morphism evident in the eval1 function is read, which describes how the value of a variable is obtained. The monadic interpreter for expressions
takes an index mapping names to locations and returns a computation producing an integer.

(* eval1 : Exp -> index -> int M *)
fun eval1 exp index =
case exp of
  | Constant n => Return mswo n
  | Variable x => let val loc = position x index
                 in read loc end
  | Minus(x,y) =>
    Do mswo { a <- eval1 x index ;
              b <- eval1 y index;
              Return mswo (a - b) }
  | Greater(x,y) =>
    Do mswo { a <- eval1 x index ;
              b <- eval1 y index;
              Return mswo (if a '>' b then 1 else 0) }
  | Times(x,y) =>
    Do mswo { a <- eval1 x index ;
              b <- eval1 y index;
              Return mswo (a * b) };

  The interpreter for Com uses the non-standard morphisms write, push, and pop to transform the stack and the morphism output to add to the output stream.

(* interpret1 : Com -> index -> unit M *)
fun interpret1 stmt index =
case stmt of
  | Assign(name,e) =>
    let val loc = position name index
     in Do mswo { v <- eval1 e index ; write loc v } end
  | Seq(sl,s2) =>
    Do mswo { x <- interpret1 s1 index;
              y <- interpret1 s2 index;
              Return mswo () }
  | Cond(e,sl,s2) =>
    Do mswo { x <- eval1 e index;
              if x=1
               then interpret1 sl index
               else interpret1 s2 index }
  | While(e,body) =>
    let fun loop () =
      Do mswo { v <- eval1 e index ;
                if v=0 then Return mswo ()
                else Do mswo { interpret1 body index ;
                               loop () } }
    in loop () end
  | Declare(nm,e,stmt) =>
    Do mswo { v <- eval1 e index ;
              push v ;
              interpret1 stmt (nm::index);
              pop }
  | Print e =>
    Do mswo { v <- eval1 e index;
              output v };

Although interpret1 is fairly standard, we feel that two things are worth pointing out. First, the clause for the Declare constructor, which calls push and pop, implicitly
changes the size of the stack and explicitly changes the size of the index (nm:index), keeping the two in sync. It evaluates the initial value for a new variable, extends the index with the variables name, and the stack with its value, and then executes the body of the Declare. Afterwards it removes the binding from the stack (using pop), all the while implicitly threading the accumulated output. The mapping is in scope only for the body of the declaration.

Second, the clause for the While constructor introduces a local tail recursive function loop. This function emulates the body of the while. It is tempting to control the recursion introduced by the While by using the recursion of the interpreter function itself by using a clause something like:

```plaintext
| While(e, body) =>
  Do mswo { v <- eval1 e index ;
    if v=0 then Return mswo ()
    else Do mswo { interpreti body index ;
                     interpreti (While(e, body)) index } }
```

Here, if the test of the loop is true, we run the body once (to transform the stack and accumulate output) and then repeat the whole loop again. This strategy, while correct, will have disastrous results when we stage the interpreter, as it will cause the first stage to loop infinitely.

There are two recursions going on here. First the unfolding of the finite data structure which encodes the program being compiled, and second, the recursion in the program being compiled. In an unstaged interpreter a single loop suffices. In a staged interpreter, both loops are necessary. In the first stage we only unfold the program being compiled and this must always terminate. Thus we must plan ahead as we follow our three step process. Nevertheless, despite the concessions we have made to staging, this interpreter is still clear, concise and describes the semantics of the while-language in a straight-forward manner.

### 4.4 Step 2: staged interpreter

To specialize the monadic interpreter to a given program we add two levels of staging annotations. The result of the first stage is the intermediate code, that if executed returns the value of the program. The use of the bracket annotation enables us to describe precisely the code that must be generated to run in the next stage. Escape annotations allow us to escape the recursive calls of the interpreter that are made when compiling a while-program.

```plaintext
(* eval2: Exp -> index -> int M *)
fun eval2 exp index =
  case exp of
  Constant n => <Return mswo "(lift n)"
  Variable x =>
    let val loc = position x index
    in <read "(lift loc)" end
  Minus(x, y) =>
    <Do mswo { a <- "(eval2 x index) ;
                 b <- "(eval2 y index);
                 Return mswo (a - b) }>
  Greater(x, y) =>
    <Do mswo { a <- "(eval2 x index) ;
                 b <- "(eval2 y index);
                 Return mswo (a - b) }>
```

8
Return mswo (if a '>' b then 1 else 0) >>
| Times(x,y) =>
| <Do mswo { a <- "(eval2 x index) ;
b <- "(eval2 y index);
Return mswo (a * b) }>;

The lift operator inserts the value of loc as the argument to the read action. The value of loc is known in the first-stage (compile-time), so it is transformed into a constant in the second-stage (run-time) by lift.

To understand why the escape operators are necessary, let us consider a simple example: eval2 (Minus(Constant 3,Constant 1)) []. We will unfold this example by hand below:

\[
\text{eval2} \ (\text{Minus} (\text{Constant} \ 3, \text{Constant} \ 1)) \ [] = \\
< \text{Do} \ \text{mswo} \\
\{ \ a <- "(\text{eval2} \ (\text{Constant} \ 3) \ []) ;
\quad \ b <- "(\text{eval2} \ (\text{Constant} \ 1) \ []) ;
\quad \ \text{Return} \ \text{mswo} \ (a - b) \} > = \\
< \text{Do} \ \text{mswo} \\
\{ \ a <- "<\text{Return} \ \text{mswo} \ 3> ;
\quad \ b <- "<\text{Return} \ \text{mswo} \ 1> ;
\quad \ \text{Return} \ \text{mswo} \ (a - b) \} > = \\
< \text{Do} \ \text{mswo} \\
\{ \ a <- \text{Return} \ \text{mswo} \ 3 ;
\quad \ b <- \text{Return} \ \text{mswo} \ 1 ;
\quad \ \text{Return} \ \text{mswo} \ (a - b) \} > = \\
< \text{Do} \ %\text{mswo} \\
\{ \ a <- \text{Return} \ %\text{mswo} \ 3 ;
\quad \ b <- \text{Return} \ %\text{mswo} \ 1 ;
\quad \ \text{Return} \ %\text{mswo} \ (a % - b) \} >
\]

Each recursive call produces a bracketed piece of code which is spliced into the larger piece being constructed. Recall that escapes may only appear at level-1 and higher. Splicing is axiomatized by the reduction rule: "<x> \rightarrow x", which applies only at level-1. The final step, where mswo and - become %mswo and %-, occurs because both are free variables and are lexically captured.

Now we can state the equivalence relationship between the monadic eval1 and the staged eval2. We use the axiomatic semantics of METAML [28], in particular the axioms for the annotations, such as the splice axiom above.

**Proposition 1.** For all expressions exp, and list of names index:

\[
\text{eval1} \ \text{exp} \ \text{index} = \ \text{run} \ (\text{eval2} \ \text{exp} \ \text{index})
\]

**Proof.** We might argue that there is a trivial proof to this proposition. Since eval1 is simply a copy of eval2 with all the staging annotations erased, and that both functions type-check, by the semantics of METAML they must be equal. We include a more traditional proof in the appendix using the axiomatic semantics of METAML [28] (see appendix A).
Interpreter for Commands.

Staging the interpreter for commands proceeds in a similar manner:

(* interpret2 : Com -> index -> 'unit H *)
fun interpret2 stmt index =
case stmt of
Assign(name,e) =>
  let val loc = position name index
  in <Do mswo { n <- (eval2 e index);
    write (lift loc) n }>
  end
| Seq(s1,s2) =>
  <Do mswo { x <- (interpret2 s1 index);
    y <- (interpret2 s2 index);
    Return mswo () }>
| Cond(e,s1,s2) =>
  <Do mswo { x <- (eval2 e index);
    if x=1
      then (interpret2 s1 index)
      else (interpret2 s2 index) }>
| While(e,body) =>
  <let fun loop () =
    Do mswo { v <- (eval2 e index);
      if v=0
        then Return mswo ()
        else Do mswo { q <- (interpret2 body index); loop ()} }
  in loop () end>
| Declare(nm,e,stmt) =>
  <Do mswo { x <- (eval2 e index);
    push x ;
    (interpret2 stmt (nm::index)) ;
    pop }>
| Print e =>
  <Do mswo { x <- (eval2 e index);
    output x }>

4.4.1 An example.

The function interpret2 generates a piece of code from a Com object. To illustrate this we apply it to the simple program: declare x = 10 in { x := x - 1; print x } and obtain:

<Do %mswo
{ a <- Return %mswo 10
 ; %push a
 ; Do %mswo
  { e <- Do %mswo
    { d <- Do %mswo
      { b <- %read 1
        ; c <- Return %mswo 1
        ; Return %mswo b % c
      }
      ; %write 1 d
    }
  ; g <- Do %mswo
}
Note that the staged program is essentially a compiler, translating the syntactic representation of the while-program into the above monadic object-program that will compute its meaning. This program sequentializes the decrement \( x \) and the print of \( x \). This object-program is fully executable. Simply by using the run operator of METAi, it can be executed for prototyping purposes.

Equally important, the object-program itself is just a piece of data, which can be analyzed and further translated in another layer of the translation pipeline. The reader might notice that this object-program could be further simplified by applying the monad laws. There are many opportunities for doing so. After these laws are applied we obtain the much more satisfying:

<Do \( \%\text{mswo} \)

\[
\{ \%\text{push} \ 10 \\
\%\text{read} \ 1 \\
\%\text{write} \ 1 \ b \\
\%\text{read} \ 1 \\
\%\text{output} \ d \\
\%\text{mswo} \}
\]

In addition to the monad laws which hold for all monads, we can also use laws which hold for particular non-standard morphisms. For instance, in the example above, we could avoid the second read of location 1 using the following rule:

\[
\text{Do} \{ \text{el}; \ c \leftarrow \%\text{write} \ 1 \ b ; \ d \leftarrow \%\text{read} \ 1 ; \ e2 \} = \text{Do} \{ \ e; \ c \leftarrow \%\text{write} \ 1 \ b ; \ e2[b/d] \}
\]

Every target language will have many such laws, and because our target language is both executable-code, and data-structure we can perform these optimizations. How this is accomplished is the subject of Section 4.5.

As for the eval function, we state the semantic equivalence between the monadic and the staged interpreters.

**Proposition 2.** For all commands \( \text{com} \) and list of names \( \text{index} \):

\[
\text{interpret1} \ \text{com} \ \text{index} = \text{run} \ (\text{interpret2} \ \text{com} \ \text{index})
\]

**Proof.** See appendix A.

### 4.5 Step 3: optimizing target code: the monadic laws

Perhaps the most important contribution we make in this paper, is that a staged program produces a piece of code that is both an executable-program and a data-structure.

If one wants to execute this code, one uses the run annotation. If one wants to optimize this code, this is possible as well. In this section we illustrate this by example; providing an implementation of the monad law transformations demonstrated in section 4.4.1
In this section, we briefly explain our method for analysing (or computing over, or
doing intensional analysis of) METAML code. We believe, that operations such as pattern-
matching and substitution on code should be provided once and for all by the system, and
not by the user.

Optimizations are generally thought of as rewriting rules or transformations. Both the
rules and the strategy (e.g. top-down or bottom-up) needed to apply them need to be
described.

To illustrate this point, we write a simple transformation which implements the monadic
laws as directed rewrites. As a reminder, the monadic laws expressed in terms of METAML’s
\( \text{Do} \) and \( \text{Return} \) notation are repeated.

\[
\text{Do} \{ x <- \text{return } e ; z \} = z[e/x]
\]
\[
\text{Do} \{ x <- m ; \ \text{return } x \} = m
\]
\[
\text{Do} \{ x <- \text{Do} \{ y <- a ; b \} ; c \} = \text{Do} \{ y' <- a ; \text{Do} \{ x <- b[y'/y] ; c \} \}
\]

To implement these rules, we need a mechanism for pattern matching over code. Like
all METAML code, the result of the monadic interpreter is just a data structure so this is
possible.

Let us consider a simple example. Suppose we want to match all pieces of code that
are of the form \( \langle x + 3 \rangle \). We have used the \( \Delta \) to indicate a meta-variable that will match
any piece of code. We cannot put a variable (e.g. \( x \)) here because \( \langle x+3 \rangle \) is just a piece
code and not a pattern. The solution to indicating a meta-variable in a pattern is to use an
escaped variable at level-1 in the pattern. Thus the pattern \( \langle \sim x + 3 \rangle \) matches all pieces
of code that have this “shape”.

Unfortunately, this scheme is not always sufficient when matching against code with
binding constructs such as \( \langle \text{fn } x => x + 1 \rangle \). We would like to construct a pattern that
matches against a function (or other binding construct) and to be able to use the meta-
variables bound inside the pattern to construct a transformation. To see why this is
problematic consider the following two examples:

1. We want a transformation that increments the body of an integer valued function,
such that when applied to \( \langle \text{fn } x => x \rangle \) we obtain \( \langle \text{fn } x => x + 1 \rangle \), and when
applied to \( \langle \text{fn } y => \text{length } y \rangle \) we obtain \( \langle \text{fn } y => (\text{length } y) + 1 \rangle \). As a first
approximation we try: \( \langle \text{fn } x => \Delta \rangle => \langle \text{fn } x => \Delta +1 \rangle \). This looks promising,
but what would happen if we wrote: \( \langle \text{fn } x => \Delta \rangle => \langle \text{fn } y => \Delta +1 \rangle \) instead?
Now, free occurrences of \( x \) in \( \Delta \) no longer have a binding site, because they have
been spliced into a context where \( y \) is the bound variable instead of \( x \).

2. We want a transformation that doubles the argument of an \( \text{int } -> \text{int} \) function,
such that when applied to \( \langle \text{fn } x => x \rangle \) we obtain \( \langle \text{fn } x => x + x \rangle \) and when
applied to \( \langle \text{fn } x => y + x \rangle \) we obtain \( \langle \text{fn } x => y + (x + x) \rangle \). The problem here
is that in the pattern, \( \langle \text{fn } x => \Delta \rangle \), there is no way to express that \( \Delta \) may have
free occurrences of \( x \) inside, and that our transformation needs to substitute for
those free occurrences.

The solution is to use a higher-order pattern. Suppose we could parameterize \( \Delta \) on \( x \).
This makes \( (\Delta, x) \) not a meta-variable with type code, but a meta-variable with type code
to code. Inside a pattern on the left hand side of a match ( \( \text{pat } => \text{exp} \) ) a higher order
meta-variable is bound to a function when it is successfully matched against a piece of
code. On the right hand side of the match, when this meta-variable is used (by applying
it to a piece of code) it substitutes all occurrences of \( x \) with the argument it was applied
to. For example consider the table below showing the binding of the higher order meta
variable \( \Delta_x \) when the pattern \( \langle \text{fn } x => \Delta_x + 3 \rangle \) is matched against different pieces of
code.
code matched against function bound to

<fn x => x + 3> fn x => \( x \)

<fn x => (x - 9) + 3> fn x => \( x - 9 \)

<fn x => (\( \sin x + x^2 \)) + 3> fn x => \( \sin x + x^2 \)

<fn x => x + 1> match failure

To express this in META\(ML\) we use the convention that the function in an escaped application (where all the arguments of the application are explicitly bracketed code) represents a higher order meta-variable. Thus, whenever an escaped application appears inside a pattern, the function part of the application is a higher-order meta-variable and its arguments are its formal parameters. For example: \( (g \ <x>)\). The two problematic examples above are now easily expressed as:

<fn x => \( (g \ <x>) \) => fn y => \( (g \ <y>) + 1 \)

<fn x => \( (h \ <x>) \) => fn z => \( (g \ <z + z>) \)

Because higher order meta-variables may appear only in the function position of escaped applications, and the arguments of these escaped applications may only be bracketed bound variables (like \( <x> \)), pattern-matching and unification are decidable [16, 25].

We now possess the tools to present the monad-law and code-optimizing META\(ML\)-function opt:

\[
\begin{align*}
\text{fun opt} & \langle \text{Do } \text{st} \{ x \leftarrow \ldots ; \text{return } \text{st} \ x \} \rangle = \text{opt } v \\
& | \ \text{opt } w \text{ as } \langle \text{Do } \text{st} \{ x \leftarrow \text{Return } \text{st} \ e ; \ (z \ <x>) \} \rangle = \\
& \ \text{if } \text{is_constant } e \ \text{then } \text{opt } (z \ e) \ \text{else } w \\
& | \ \text{opt} \langle \text{Do } \text{st} \{ x \leftarrow \text{Do } \text{st'} \{ y \leftarrow \ldots ; \ (f \ <y>) \} ; \ (g \ <x>) \} \rangle = \\
& \ \text{opt} \langle \text{Do } \text{st} \{ y' \leftarrow \ldots ; x' \leftarrow \ (f \ <y'>) ; \ (g \ <x'>) \} \rangle \\
& | \ \text{opt } x = \text{map_code opt } x \quad (* \text{traversal through the code} *)
\end{align*}
\]

Our opt function implements a limited form of the left-id monad law. We do not wish to duplicate by substitution a non-constant. By composing this optimization with interpret2 we obtain a better compiler. Applying this compiler to:

```
Declare x = 150 in
  Declare y = 200 in while x > 0 Do { x := x - 1; y := y - 1 }
```

we obtain following program:

```
<Do %state
  \{ a <- %push 150;
    b <- %push 200;
    c <- let fun loop () =
      Do %state
        \{ e <- %read 1;
          f <- return %state (if (e %> 0) then 1 else 0);
          if (f %= 0)
            then return %state ()
            else Do %state
              \{ g <- %read 1;
                h <- return %state (g - 1);
                i <- %write 1 h;
```
The optimizer has fully sequentialized the code using the bind-associativity law, and removed all superfluous Return's using the unit-identity laws. Further optimizations, such as arithmetic simplification, or transformations to another form, such as assembly code, could be implemented in the same fashion.

5 Related work

Our work was inspired by work in many different areas. Derivation of compilers from specifications and the use of action-semantics [19, 23, 11, 22]; the use of monads to structure programs in general [18, 31, 26] and language implementations in particular [32, 27, 14]; staged programming [5, 6] and its use in structuring compilers [29, 20, 4]; partial evaluation [34, 17, 1, 3, 2, 9]; higher order abstract syntax and pattern matching [16, 7].

For space considerations we limit detailed discussion to the following areas.

5.1 Monads and compilation

Perhaps, the most related work is the work of Sheng Liang and his thesis advisor Paul Hudak [12, 13]. They investigate the derivation of a compiler from a modular monadic interpreter. Our work is a continuation of their effort of using monads as a standard compilation mechanism. However, some differences remain:

- The use of staging, lead us at an early step in the development, to split the environment into a static index of names and a dynamic stack of values. This allows us to avoid the use of an environment monad. We use instead an state transformer monad in which the state is managed like a stack. Liang uses a complicated monad which is a combination of an environment monad and a state transformer. After code generation they show that the residual code due to the environment (the lookups of the location of variables) can be eliminated using axioms of the non-standard morphisms of the environment monad. Our use of staging allowed us to do the lookups in the first stage and to never residualize the lookups at all.

- On the other hand, Liang's use of modular language components is an advantage we have not even attempted to employ. For simplicity, we have used the same monad for both expressions and commands while Liang uses a modular approach where each feature is defined independently from the others. Finally all the features are combined by a monad transformer. To do this it is necessary to lift all non-standard morphisms through the transformer. This is hard and not completely understood. We may try to duplicate Liang's approach in future work.
5.2 Staging and compilation

In his thesis *Calculating Compilers* [15] Erik Meijer advocates staging a compiler by using self discipline. Construct a compiler by building it as the composition of compile-time and run-time components. A critical step in this process is finding a representation of every source language construct as a combination of (lower level) target level constructs. By representing both source and target languages as algebraic datatypes, say source and target, induced by the functors S and T, this can be reduced to finding a polymorphic function Trans, which for all \( \alpha \), has type \((T\alpha \rightarrow \alpha) \rightarrow (S\alpha \rightarrow \alpha)\), a so-called algebra transformer.

Let the semantic domain of the target algebra be some type value. If the semantic meaning function for the target language \( M: \text{target} \rightarrow \text{value} \) can be expressed as a catamorphism \( M = \text{cata} \phi \) where \( \phi: \text{T value} \rightarrow \text{value} \), we can lift \( \phi \) into an interpreter for the source language by applying the algebra transformer Trans. Thus \( \text{Trans} \phi: \text{S value} \rightarrow \text{value} \) and \( \text{Interp} = \text{cata} (\text{Trans} \phi): \text{source} \rightarrow \text{value} \). A similar construction can be used to construct the compiler \( \text{Compiler}: \text{source} \rightarrow \text{target} \).

Let function \( \text{In}: \text{T target} \rightarrow \text{target} \) be the injection between the functor T and its induced algebraic datatype target, then \( \text{cata} (\text{Trans} \text{In}): \text{source} \rightarrow \text{target} \) constructs the compiler.

The limiting factor in this approach is finding an algebraic datatype target to encode the target language. For a monadic target language, it is not known how to do this, since the constructors for “unit” and “bind” would be too polymorphic to encode in an algebraic datatype, and many of the non-standard morphisms would not be polymorphic enough.

By staging the process in METAML, we do away with the need for an algebraic datatype to encode the target language, by using the special type of code instead. The constructors of the target algebra are simply the second stage representations of the real functions.

5.3 Difference between staging and partial evaluation

Staged programming (S.P.) is closely related to partial evaluation (P.E.). We list what we believe are the salient differences.

- S.P. uses explicit annotations while P.E. uses implicit annotations placed by an automatic binding time analysis.
- S.P. gives the programmer complete control over what residual program is produced, while the residual program produced by P.E. often contains surprises. The surprises are caused by the differences between what the programmer knows and what the binding time analysis can discover. The solution to this mismatch is for the programmer to restructure his program using “binding-time improvements” which more closely align his knowledge and the capabilities of the binding time analysis. Of course S.P. is not completely immune to these difficulties, but the staged programmer must be fully aware of the staging issues before he writes his program. The staged type-system is a great advantage here. Nevertheless, there are many simple programs where automatic binding time analysis is sufficient, and hand staging is simply an annoyance. In our system we have combined the advantages of both, allowing a simple type-directed binding time analysis to co-exist with the manual staging annotations. An analysis of this co-existence is beyond the scope of this paper.
- S.P. is a programming language feature. It exists at the same level as the program. Here the algorithm and the staging are developed hand in hand. There are no
additional tools or processes, and users learn how to weave the staging thought processes into their problem solving techniques.

- S.P. provides a complete, unified, typed environment, supporting both type reconstruction and polymorphism for the staged constructs.

6 The Implementation

Everything you have seen in this paper, except the higher order pattern matching over code, has been implemented in the METAML implementation. The examples are actual runs of the system.

The higher order pattern matching is currently under development. We found the normalizing effect of the monad laws\(^1\) so compelling that we implemented them in an ad-hoc fashion inside the METAML system.

7 Conclusion

We have shown that staging programs offers an exciting new programming paradigm, and reinforced the notion that staging a monadic interpreter into compile-time and run-time components provides a direct link between an interpreter and a compiler.

Acknowledgements. The research reported in this paper was supported by the USAF Air Materiel Command, contract #F19628-93-C-0069, and NSF Grant IRI-9625462.

References


\(^1\)as well as the laws for $\beta$-value, let-normalization, and $\eta$-value reduction


A Proofs

We repeat here the axiomatic semantics of METAML [28]. For the sake of simplicity, we omit the level-annotations.

\[
\begin{align*}
\text{run } \langle v1 \rangle &= v1 \downarrow \\
\text{run } \langle e \rangle &= e \\
\text{(escape)} \\
\text{run } (\lambda x. e) v &= e[x := v] \\
\text{(beta)}
\end{align*}
\]

The (escape) axiom applies only at level one (inside exactly one bracket) and (run) and (beta) apply only at level 0 (inside no brackets).

Lemma 1. For any well-typed expression: \( \langle \neg e \rangle \), we have \( \langle \neg e \rangle = e \)

Proof. Since \( \langle \neg e \rangle \) is well-typed, \( e \) must evaluate (if it terminates) to \( \langle v \rangle \). Then \( e = \langle v \rangle \). We have

\[
\begin{align*}
\langle \neg e \rangle &= \langle \neg \langle v \rangle \rangle \\
&= \langle v \rangle & \text{replace equals by equals} \\
&= e & \text{By escape axiom}
\end{align*}
\]

Lemma 2. For any well-type expression: run \( \langle f e \rangle \), we have

\[\text{run } \langle f e \rangle = (\text{run } \langle f \rangle) (\text{run } \langle e \rangle).\]

Proof. Since the term \( f e \) is at level 1, the only possible reduction is by the escape axiom. Assume \( \langle f \rangle \) and \( \langle e \rangle \) evaluate to the values \( \langle f_1 \rangle \) and \( \langle e_1 \rangle \) respectively. Then \( \langle f e \rangle \) must evaluate to \( \langle f_1 e_1 \rangle \) (since at level 1 we cannot do a beta-step). Hence, we have \( \langle e \rangle = \langle e_1 \rangle, \langle f \rangle = \langle f_1 \rangle, \langle f e \rangle = \langle f_1 e_1 \rangle \)

\[
\begin{align*}
\text{run } \langle f e \rangle &= \text{run } \langle f_1 e_1 \rangle \\
&= (\text{run } \langle f_1 \rangle) (\text{run } \langle e_1 \rangle) & \text{by replacing equals by equals} \\
&= (f_1 e_1) \downarrow & \text{by run axiom} \\
&= (\text{run } \langle f_1 \rangle) (\text{run } \langle e_1 \rangle) & \text{by definition of } \downarrow \\
&= (\text{run } \langle f \rangle) (\text{run } \langle e \rangle) & \text{by replacing equals by equals}
\end{align*}
\]

Lemma 3. For any well-type expression: run \( \langle \lambda x. e \rangle \), we have

\[\text{run } \langle \lambda x. e \rangle = \lambda x. (\text{run } \langle e \rangle).\]

Proof. The proof is similar to the two lemmas above.

A consequence of the previous two lemmas is that \text{run} distributes through its subexpressions. In particular, \text{run} distributes through Do and let.

\[
\begin{align*}
\text{run } \text{Do} \{ x_1 \leftarrow e_1 ; x_2 \leftarrow e_2 ; \ldots ; \text{en} \} &= \\
\text{Do} \{ x_1 \leftarrow \text{run } \langle e_1 \rangle ; x_2 \leftarrow \text{run } \langle e_2 \rangle ; \ldots ; \text{run } \langle \text{en} \rangle \} & \text{(run-Do)}
\end{align*}
\]

\[
\begin{align*}
\text{run } \text{let } v \text{a}l x = e \text{ in } e_2 &= (\text{let } v \text{a}l x = \text{run } \langle e_1 \rangle \text{ in } \text{run } \langle e_2 \rangle) & \text{(run-Let)}
\end{align*}
\]

Proposition 1. For all expressions \( \text{exp} \), and list of names \( \text{index} \):

\[
\text{eval1 } \text{exp } \text{index} = \text{run } (\text{eval2 } \text{exp } \text{index})
\]
Proof. Induction on the structure of \( \text{exp} \).

**case exp of Minus\((e_1,e_2)\)**

\[
\text{run}\ (\text{eval2} \ (\text{Minus} (e_1,e_2)))\ \text{index} = \text{By beta axiom}
\]

\[
\text{run} <\text{Do msw}\ \{\ a \leftarrow (\text{eval2} \ e_1 \ \text{index}); \ b \leftarrow (\text{eval2} \ e_2 \ \text{index}); \ \text{Return msw} \ (a-b) \}> = \text{by (run-Do)}
\]

\[
\text{Do msw} \ \{\ a \leftarrow \text{run} <(\text{eval2} \ e_1 \ \text{index})>; \ b \leftarrow \text{run} <(\text{eval2} \ e_2 \ \text{index})>; \ \text{run} <\text{Return msw} \ (a-b) \} = \text{by lemma 1 (twice) and run axiom}
\]

\[
\text{Do msw} \ \{\ a \leftarrow \text{run} \ (\text{eval2} \ e_1 \ \text{index}); \ b \leftarrow \text{run} \ (\text{eval2} \ e_2 \ \text{index}); \ \text{Return msw} \ (a-b) \} = \text{by induction hypothesis (twice)}
\]

\[
\text{eval1} \ (\text{Minus} (e_1,e_2))\ \text{index}
\]

The other cases are similar. \( \square \)

**Proposition 2.** For all commands \( \text{com} \) and list of names \( \text{index} \):

\[
\text{interpret1} \ \text{com} \ \text{index} = \text{run} \ (\text{interpret2} \ \text{com} \ \text{index})
\]

**Proof.** By induction on the structure of \( \text{com} \).

**case com of While\((e,\text{body})\).**

\[
\text{run} \ (\text{interpret2} \ (\text{While} (e,\text{body})))\ \text{index} = \text{By beta}
\]

\[
\text{run} <\text{let fun loop () =}
\text{Do msw} \ \{\ v \leftarrow (\text{eval2} \ e \ \text{index}); \ \text{if} \ v=0 \text{then Return msw} () \text{else Do msw} \ \{\ q \leftarrow (\text{interpret2} \ \text{body} \ \text{index}); \ \text{loop ()}\} \}
\text{in loop () end} > = \text{by run-Do and run-Let}
\]

\[
\text{let fun loop () =}
\text{Do msw} \ \{\ v \leftarrow \text{run} <(\text{eval2} \ e \ \text{index})>; \ \text{if} \ v=0 \text{then run} <\text{Return msw} ()> \text{else Do msw} \ \{\ q \leftarrow \text{run} <(\text{interpret2} \ \text{body} \ \text{index})>; \ \text{run} <\text{loop ()}>\}
\text{in run <loop ()> end} = \text{By Lemma 1 and run axiom}
\]
let fun loop () =
  Do msw0 { v <- run (eval2 e index)>;
    if v=0
      then Return msw0 ()
    else Do msw0 { q <- run (interpret2 body index);
                   run < loop () >}
  }
  in run < loop ()> end  = By induction hypothesis and Proposition 1

let fun loop () =
  Do msw0 { v <- (eval1 e index)>;
    if v=0
      then Return msw0 ()
    else Do msw0 { q <- interpret1 body index); run <loop ()> }
  }
  in run <loop ()> end  = By run axiom

let fun loop () =
  Do msw0 { v <- (eval1 e index)>;
    if v=0
      then Return msw0 ()
    else Do msw0 { q <- interpret1 body index); loop () }
  }
  in loop () end

The last step is only possible because, at this step in the derivation, there are no annotations (in particular no escapes) in the body of the function loop, thus the body of loop at level 1 is a value, and hence in normal form.

The other cases are easier.  □
Multi-Stage Programming: Axiomatization and Type Safety
extended abstract

Walid Taha & Zine-El-Abidine Benaissa & Tim Sheard
January 14, 1998

Abstract

Multi-staged programming provides a new paradigm for constructing efficient solutions to complex problems. Techniques such as program generation, multi-level partial evaluation, and run-time code generation respond to the need for general purpose solutions which do not pay run-time interpretive overheads. This paper provides a foundation for the formal analysis of one such system.

We introduce a multi-stage language and present its axiomatic, reduction, and natural semantics. Our axiomatic semantics is an extension of the call-by-value λ-calculus with staging constructs. We demonstrate the soundness of the axiomatic semantics with respect to the natural semantics. We show that staged-languages can "go Wrong" in new ways, and devise a type system that screens out such programs. Finally, we present a proof of the soundness of this type system with respect to the reduction semantics, and show how to extend this result to the natural semantics.

1 Introduction

Recently, there has been significant interest in various forms of multi-stage computation, including program generation [3, 26], multi-level partial evaluation [11, 12], and run-time code generation [1, 5, 4, 8, 9, 13, 15, 16, 22]. Such techniques combine both the software engineering advantages of general purpose systems and the efficiency of specialized ones.

Because such systems execute generated code *never inspected by human eyes* it is important to use formal analysis to guarantee properties of this generated code. We would like to guarantee statically that a program generator synthesizes only programs with properties such as: type-correctness, global references only to names in scope, and local names which do not inadvertently hide global references.

In previous work [25], we introduced a multi-stage programming language called MetaML. In that work we introduced four staging annotations to control the order of evaluation of terms. We argued that staged programs are an important mechanism for constructing general purpose systems with the efficiency of specialized ones, and addressed engineering issues necessary to make such systems usable by programmers. We introduced an operational semantics and a type system to screen out bad programs, but we were unable to prove the soundness of the type system.

Further investigation revealed important subtleties that were not previously apparent to us. In this paper, we report on work rectifying some of the practical limitations of our previous work. In contrast to our earlier work that focused on implementations and problem solving using multi-staged programs, this paper reports on a more abstract treatment of MetaML's foundations. The key results reported in this paper are as follows:

1. An axiomatic semantics and a reduction semantics for a core sub-language of MetaML.
2. A characterization of the additional ways in which a staged program can "go Wrong".
3. A type system to screen out such programs.

4. A soundness proof for the type system with respect to the reduction semantics using the syntactic approach to type-soundness of Wright and Felliesen [27].

5. A natural semantics that chooses the order in which rules are applied.

6. The soundness of the axiomatic semantics with respect to the natural semantics.

These results form a strong, tightly-woven foundation which gives us both a better understanding of MetaML, and more confidence in the well-foundedness of the multi-stage paradigm. The axiomatic semantics provides us with an equational theory for formally reasoning about the equivalence of MetaML programs, and the reduction semantics is an abstract characterization of the notion of staged computation. The natural semantics provides us with a deterministic strategy for implementing multi-stage computation. The soundness of the axiomatic semantics with respect to the natural semantics formally demonstrates that results based on the reductions semantics are also applicable to our implementation. Finally, formally proving the soundness of the type system with respect to the reduction semantics ensures to us that well-typed programs are well-behaved.

1.1 What are Staged Programs All About?

In staging a program, the user has control over the order of evaluation of terms. This is done by using staging annotations. In MetaML the staging annotations are Brackets $<$, Escape $\_\_$ and run. An expression $<$e$>$ defers the computation of e; $\_\_$ splices the deferred expression obtained by evaluating e into the body of a surrounding Bracketed expression; and run e evaluates e to obtain a deferred expression, and then evaluates this deferred expression. It is important to note that $\_\_$ is only legal within lexically enclosing Brackets. To illustrate, consider the script of a small MetaML session below:

```
-| val pair = (3+4,<3+4>);
val pair = (7,<3+4>) : (int * <int>)

-| fun f (x,y) = < 8 - ~y >;
val f = fn : ('a * <int>) -> <int>

-|  val code = f pair;
val code = <8 - (3+4)>  :  <int>

-|  run code;
val it = 1  :   int
```

The first declaration defines a variable pair. The first component of the pair is evaluated, but the evaluation of the second component is deferred by the Brackets. Brackets in types such as $<$int$>$ are read “Code of int”, and distinguish values such as $<$3+4$>$ from values such as 7. The second declaration illustrates that code can be abstracted over, and that it can be spliced into a larger piece of code. The third declaration applies the function f to pair performing the actual splicing. And the last declaration evaluates this deferred piece of code.

To give a brief feel for how MetaML is used to construct larger pieces of code at run-time consider:

```
-| fun mult x n = if n=0 then $<$1$>$ else $\_\_$ (mult x (n-1)) >;
val mult = fn : <int> -> int -> <int>
```
The function \( \text{mult} \), given an integer piece of code \( x \) and an integer \( n \), produces a piece of code that is an \( n \)-way product of \( x \). This can be used to construct the code of a function that performs the cube operation, or generalized to a generator for producing an exponentiation function from a given exponent \( n \). Note how the looping overhead has been removed from the generated code. This is the purpose of program staging and it can be highly effective as discussed elsewhere [4, 10, 13, 17, 22, 25]. In this paper we move away from how staged languages are used and address their foundations.

2 The \( \lambda \)-R Language

The \( \lambda \)-R language represents the core of MetaML. It has the following syntax:

\[ e ::= i \mid x \mid e e \mid \lambda x . e \mid \langle e \rangle \mid \langle e \rangle \mid \text{run } e \]

which includes the normal constructs of the \( \lambda \)-calculus, integer constants, and the three additional staging constructs.

To define the semantics of Escape, which is dependent on the surrounding context, we choose to explicitly annotate all terms with their level. The level of a term is the number of Brackets minus the number of Escapes surrounding that term. We define level-annotated terms as follows:

\[
\begin{align*}
    a^0 & ::= i^0 \mid x^0 \mid (a^0 a^0)^0 \mid (\lambda x . a^0)^0 \mid <a^1>^0 \mid (\text{run } a^0)^0 \\
    a^{n+1} & ::= i^{n+1} \mid x^{n+1} \mid (a^{n+1} a^{n+1})^{n+1} \mid (\lambda x . a^{n+1})^{n+1} \mid <a^{n+2}>^{n+1} \mid (-a^n)^{n+1} \mid (\text{run } a^{n+1})^{n+1}
\end{align*}
\]

Note that Escape never appears at level 0 in a level-annotated term. We define a \( \lambda \)-R program as a closed term \( a^0 \). Hence, example programs are \( (\lambda x . x^0)^0 \) and \( ((\lambda x . (x^2 x^2)^2)^2 5^2)^2>^1>^0 \).

2.1 Values

It is instructive to think of values as the set of terms we consider to be acceptable results from a computation. Values are defined as follows:

\[
\begin{align*}
    v^0 & ::= i^0 \mid x^0 \mid (\lambda x.a)^0 \mid <v^1>^0 \\
    v^1 & ::= i^1 \mid x^1 \mid (v^1 v^1)^1 \mid (\lambda x.v^1)^1 \mid <v^2>^1 \mid (\text{run } v^1)^1 \\
    v^{n+2} & ::= i^{n+2} \mid x^{n+2} \mid (v^{n+2} v^{n+2})^{n+2} \mid (\lambda x.v^{n+2})^{n+2} \mid <v^{n+3}>^{n+2} \mid (-v^{n+1})^{n+2} \mid (\text{run } v^{n+2})^{n+2}
\end{align*}
\]

The set of values for \( \lambda \)-R has three notable points. First, values can be bracketed expressions. This means that computations can return pieces of code representing other programs. Second, values can contain applications such as \((\lambda y.y^1)^1 (\lambda x.x^1)^1\). Third, there are no level 1 Escapes in values. We take advantage of this important property of values in many proofs and propositions in our present work.

Because each rule in the inductive definition above is an instance of one of the rules given in the inductive definition for level-annotated terms it is easy to show that values are a subset of level-annotated terms.

2.2 Contexts

We generalize the notion of contexts [2] to a notion of annotated contexts:
\( c^0 := [\ ] | (c^0 a^0) | (a^0 c^0) | (\lambda x.c^0) | \langle c^1 \rangle | (\text{run } c^0) \)
\( c^{n+1} := [\ ] | (c^{n+1} a^{n+1}) | (a^{n+1} c^{n+1}) | (\lambda x.c^{n+1}) | \langle c^{n+2} \rangle^{n+1} | (\text{run } c^{n+1})^{n+1} \)

where [\ ] is a hole. When instantiating an annotated context \( c^n[\ ]^m \) to a term \( e^m \) we write \( c^n[e^m] \).

### 2.3 Promotion and Demotion

The axioms of MetaML remove Brackets from level-annotated terms. To maintain the consistency of the level-annotations we need an inductive definition for incrementing and decrementing all annotations on a term. We call these operations **promotion** and **demotion**.

**Promotion**

\[
\begin{align*}
    x^n & \uparrow = x^{n+1} \\
    (a_1 a_2)^n & \uparrow = (a_1 \uparrow a_2 \uparrow)^{n+1} \\
    (\lambda x.a)^n & \uparrow = (\lambda x.a \uparrow)^{n+1} \\
    \langle a \rangle^n & \uparrow = \langle a \uparrow \rangle^{n+1} \\
    (\neg a)^n & \uparrow = (\neg a \uparrow)^{n+2} \\
    (\text{run } a)^n & \uparrow = (\text{run } a \uparrow)^{n+1} \\
    i^n & \uparrow = i^{n+1}
\end{align*}
\]

**Demotion**

\[
\begin{align*}
    x^{n+1} & \downarrow = x^n \\
    (a_1 a_2)^{n+1} & \downarrow = (a_1 \downarrow a_2 \downarrow)^n \\
    (\lambda x.a)^{n+1} & \downarrow = (\lambda x.a \downarrow)^n \\
    \langle a \rangle^{n+1} & \downarrow = \langle a \downarrow \rangle^n \\
    (\neg a)^{n+2} & \downarrow = (\neg a \downarrow)^{n+1} \\
    (\text{run } a)^{n+1} & \downarrow = (\text{run } a \downarrow)^n \\
    i^{n+1} & \downarrow = i^n
\end{align*}
\]

Promotion is a total function over level-annotated terms and is defined by a simple inductive definition. Demotion is a partial function over level-annotated terms. Demotion is undefined on terms Escaped at level 1, and on level 0 terms in general.

An important property of demotion is that while it is partial over level-annotated terms it is total over values. Proof of this is a simple induction on the structure of values.

### 2.4 Substitution

The definition of substitution is standard for the most part. In this paper we are concerned only with the substitution of values for variables. When the level of a value is different from the level of the term in which it is being substituted, promotion (or demotion, whichever is appropriate) is used to correct the level of the subterm.

\[
\begin{align*}
    i^n[x^n := v^n] & = i^n \\
    x^n[x^n := v^n] & = v^n \\
    y^n[x^n := v^n] & = y^n \\
    (a_1 a_2)^n[x^n := v^n] & = ((a_1[x^n := v^n]) (a_2[x^n := v^n]))^n \\
    (\lambda x.a)^n[x^n := v^n] & = (\lambda x.a_1)^n \\
    (\lambda y.a_1)^n[x^n := v^n] & = (\lambda y'.(a_1[y^n := y'[x^n := v^n]])^n \\
    \langle a \rangle^n[x^n := v^n] & = \langle a_1[x^{n+1} := v^n \uparrow \rangle^{n+1} \\
    (\neg a)^{n+1}[x^{n+1} := v^{n+1}] & = (\neg(a_1[x^n := v^{n+1} \downarrow]))^{n+1} \\
    (\text{run } a_1)^n[x^n := v^n] & = (\text{run } (a_1[x := v^n]))^n
\end{align*}
\]

This function is total because both promotion and demotion are total over values. A richer notion of demotion is needed to perform substitution of a variable by any expression. This generalization is beyond the scope of this paper.

### 2.5 Axiomatization and Reduction Semantics of λ-R

The axiomatic semantics describes an equivalence between two level-annotated terms. Axioms can be thought of as pattern-based equivalence rules, and are applicable in a context-independent way.
to any subterm that they match. The three axioms we will introduce can each be given a natural orientation or direction, reducing "bigger" terms to "smaller" terms. This provides a reduction semantics.

<table>
<thead>
<tr>
<th>Axiomatic</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\lambda x. e^n) v^n)^n = e^n[x := v^n]</td>
<td>((\lambda x. e^n) v^n)^n \xrightarrow{\beta} e^0[x := v^n]</td>
</tr>
<tr>
<td>((\text{run} &lt;e^{n+1}&gt;)^n = v^{n+1} \downarrow)</td>
<td>((\text{run} &lt;e^{n+1}&gt;)^n \xrightarrow{\text{run}} v^{n+1} \downarrow)</td>
</tr>
<tr>
<td>((&lt;e^{n+1}&gt;)^n + 1 = e^{n+1})</td>
<td>((&lt;e^{n+1}&gt;)^n + 1 \xrightarrow{\text{esc}} e^{n+1})</td>
</tr>
</tbody>
</table>

We write \(\lambda-R \vdash M = N\) when \(M = N\) is provable by the above axioms and the classical inference rules of an equational theory, and we write \(\rightarrow\) for the reflexive, transitive, context closure of \(\rightarrow\).

**Theorem 1 (Confluence).** *The reduction semantics is confluent.*

**Proof.** Using a notion of parallel reduction and a Strip Lemma, following closely the development in [2, pages 277–283].

**Corollary 2 (Church-Rosser).** *The axiomatic semantics is Church-Rosser.*

### 3 Faulty Terms

Under the reduction semantics, when a term has been sufficiently reduced, we would like such a term to be a *value*, but this is not always the case. If no rules apply, and the term is not a value, we say that such a term is *stuck* [27]. There are four contexts in which such terms can arise:

1. A non-\(\lambda\) value in a function position in an application (at level 0). This is the familiar form of undesirable behavior arising whenever the pure \(\lambda\)-calculus is extended with constants. For example, \(<5^1> 3^0\) is stuck because \(5^1\) is a piece of code, not a \(\lambda\)-abstraction. This term is not a value and contains no redex.

2. A variable appears at a level lower than the level at which it was bound. This is the key, distinguishing form of undesirable behavior in multi-stage computation [25]. For example: \(<(\lambda x. x^0)^1>^0\) is stuck since \(x\) is used at level 0 but bound at level 1.

3. A non-Bracket value is the argument to Run. For example: \((\text{run} 7^0)^0\) is stuck since \(7^0\) is an integer and not a piece of code.

4. A non-Bracket value is the argument to Escape. For example: \(<(4^1 + 7^0)^1>^0\)

We wish to consider as *faulty*, terms in the form above. We will show that if a term is typable, then it is not faulty, and neither can it reduce to a faulty term. We formalize this notion in the next sections.

We can now present the following formal specification for the set of faulty terms \(F\):

1. \(c[[<e^{n+1}>]^n e^n] \in F\) Non-\(\lambda\) terms in an application like: \((5^0 3^0)^0\) and \((<5^2> 3^1)^1\)

2. \(c[[\lambda x. c[z^n]]^m] \in F\) where \(m > n\). Variables at too low a level like: \(<(\lambda x. x^0)^1>^0\)

3. \(c[[\text{run} (\lambda x. e^n)]^n] \in F\) Non-Bracket in Run like: \((\text{run} (\lambda x. x)^0)^0\) and \((\text{run} 4^3)^3\)

4. \(c[[<(\lambda x. e^n)^{n+1}]] \in F\) Non-Bracket in Escape like: \(<(4^1 + (\lambda x. x)^0)^1>^0\) and \(<(4^3 + (5^2)^3)^2>^2\)
The success of our specification of faulty expressions depends on whether they help us characterize the behavior of our reduction semantics. The following lemma is an example of such a characterization, and is needed for our proof of type soundness.

**Lemma 3 (Uniform Evaluation).** Let $e^n$ be a closed term. If $e^n$ is not faulty then either it is a value or it contains a redex.

**Proof:** By induction on the structure of $e^n$.

### 4 Type System

The main obstacle to defining a sound type system for our language is the interaction between Run and Escape. While this is problematic, it adds significantly to the expressiveness of a staged language [23], so it is worthwhile overcoming the difficulty. The problem is that Escape allows Run to appear inside a Bracketed $\lambda$-abstraction, and it is possible for Run to “drop” that $\lambda$-bound variable to a level lower than the level at which it is bound. The following example illustrates the phenomenon:

$$<(\lambda x.(\tau (\text{run}<x^1>0)^0)^1)^0 \rightarrow (\lambda x.(x^0)^1)^1>$$

To avoid this problem, for each $\lambda$-abstraction we need to count the number of surrounding Runs for each occurrence of its bound variable (here $x^1$) in its body. We use this count to check that there are enough Brackets around each formal parameter to execute all surrounding Runs without leading to a faulty term.

The type system for $\lambda$-R is defined by a judgment $\Delta \vdash e^n : \tau, m$, where $e^n$ is our well-typed expression, $\tau$ is the type of the expression, $m$ is the number of the surrounding Run annotations of $e^n$ and $\Delta$ is the environment assigning types to term variables.

<table>
<thead>
<tr>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>types</td>
</tr>
<tr>
<td>type assignments</td>
</tr>
<tr>
<td>judgments</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type System</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta(x) = (\tau, j)^i</td>
</tr>
<tr>
<td>$\frac{}{\Delta \vdash x^n : \tau, m} \quad \text{Var}$</td>
</tr>
<tr>
<td>$\frac{}{\Delta \vdash i^n : \text{int}, m} \quad \text{Int}$</td>
</tr>
<tr>
<td>$\frac{}{\Delta \vdash e^n : &lt;\tau&gt;, m + 1} \quad \text{Run}$</td>
</tr>
<tr>
<td>$\frac{}{\Delta \vdash (\text{run } e^n)^n : \tau, m} \quad \text{Bra}$</td>
</tr>
<tr>
<td>$\frac{}{\Delta \vdash (\text{run } e^n)^n : &lt;\tau&gt;, m} \quad \text{Esc}$</td>
</tr>
<tr>
<td>$\frac{}{\Delta \vdash e^2_n : \tau', m} \quad \Delta \vdash e^1_i \rightarrow \tau, m \quad \text{App}$</td>
</tr>
<tr>
<td>$\frac{}{\Delta \vdash (\text{run } e^n)^n : \tau, m} \quad \text{Lam}$</td>
</tr>
</tbody>
</table>

The type system employs a number of mechanisms to reject terms that either are, or can reduce to faulty terms. The **App** rule has the standard role, and rejects non-functions applied to arguments.

The **Escape** and **Run** rules require that their operand must have type Code. This means terms such as $\text{run} 5$ and $<\lambda x.^x>$ are rejected. But while this restriction in the **Escape** and **Run** rules rejects faulty terms, it is not enough to reject all terms that can be reduced to faulty terms. The first example of such a term is $<\lambda x. (\text{run } <x>)>$ which would be typable if we use only the restrictions discussed above, but reduces to the term $<\lambda x. x>$ which would not be typable. The
second examples involves an application \((\lambda f . \lambda x . \varnothing(f \varnothing x))(\lambda x . \text{run } x)\) which would also be typable, but reduces to \(\lambda x . \varnothing\). To reject such terms we need the \textbf{Var} rule.

The \textbf{Var} rule is instrumented with the condition \(i + m \leq n + j\). Here \(i\) is the number of Bracket’s surrounding the \(\lambda\)-abstraction where the variable was bound, \(m\) is the number of Runs surrounding this occurrence of the variable, \(n\) is the number of Brackets surrounding this occurrence of the variable, and \(j\) is the number of Runs surrounding the \(\lambda\)-abstraction where it was bound. This ensures that every variable has more Brackets than Runs surrounding it.

In previous work, we have attempted to avoid these two kinds of problems using two distinct mechanisms: First, the argument of Run cannot contain free variables, and second, we prohibit the \(\lambda\)-abstraction of Run. We used unbound polymorphic type variable names in a scheme similar to that devised by Launchbury and Peyton Jones for ensuring the safety of state in Haskell [14]. It turns out that not allowing any free variables is too strong, and that using polymorphism was too weak. It is better to simply take account of the number of surrounding occurrences of Run in the \textbf{Var} rule. This way we ensure that if Run is ever in a \(\lambda\)-abstraction, it can only strip away Brackets that are explicitly apparent in that \(\lambda\)-abstraction.

5 Type Soundness of the Reduction Semantics

The type soundness proof closely follows the subject reduction proofs of Wright and Felliesen [27]. Once the reduction semantics and type system have been defined, the syntactic type soundness proof proceeds as follows:

1. Show that reduction in the standard reduction semantics preserves typing. This is called \textit{subject reduction}.
2. Show that faulty terms are not typable.

If programs are well-typed, then the two results above can be used as follows: By (1), evaluation of a well-typed program will only produce well-typed terms. By Lemma 3, every such term is either faulty, or a value, or contains a redex. The first case is impossible by (2). Thus the program either reduces to a well-typed value or it diverges.

5.1 Subject Reduction

The Subject Reduction Lemma states that a well-typed term remains well-typed under reduction. The proof relies on the Demotion, Promotion and Substitution Type Preservation Lemmas. First we need to introduce two operations on the environment assigning types to term variables:

\[
\Delta \uparrow_{(q,p)}^i (x) = (\tau, j + q)^{i+p} \text{ iff } \Delta(x) = (\tau, j)^i \\
\Delta \downarrow_{(q,p)}^i (x) = (\tau, j)^i \text{ iff } \Delta(x) = (\tau, j + q)^{i+p}
\]

These two operations map environments to environments. They are needed in the Promotion and Demotion Lemmas. They provide an environment necessary to derive a valid judgement for a promoted or demoted well-typed value. Notice that we have the following two properties:

\[
(\Delta \uparrow_{(q,p)}) \uparrow_{(i,j)} = \Delta \uparrow_{(q+i,p+j)} \text{ and } (\Delta \downarrow_{(q,p+j)}) \downarrow_{(i,j)} = \Delta \downarrow_{(q,p)}
\]

We write \(\uparrow^p\) and \(\downarrow^p\), respectively, for an abbreviation of \(p\) applications of \(\uparrow\) and \(\downarrow\) to \(v\). Note that this operation is different from \(\uparrow_{(q,p)}\) and \(\downarrow_{(q,p)}\) which is a function on environments assigning types to term variables.

\textbf{Lemma 4 (Demotion).} If \(q \leq p\) and \(\Delta_2 \downarrow_{(q,p)}\) is defined and \(\Delta_1 \cup \Delta_2 \vdash v^{n+p} : \tau, m + q\) then \(\Delta_1 \cup (\Delta_2 \downarrow_{(q,p)}) \vdash v^{n+p} \downarrow^p : \tau, m\).
Proof. By induction on the structure of \(v^{n+p}\). We develop only the variable case \(v^{n+p} = x^{n+p}\). There are only two possible sub-cases, which are:

\[
\Delta_1(x) = (\tau, j)^i \quad i + m + q \leq n + j + p
\]

\[
(\Delta_1 \cup \Delta_2) \vdash x^{n+p} : \tau, m + q \tag{Var}
\]

By hypothesis \(q \leq p\) implies \(m + i \leq n + j\). Hence \((\Delta_1 \cup (\Delta_2 \uparrow_{q,p})) \vdash v^{n+p} \downarrow^p : \tau, m\).

\[
\Delta_2(x) = (\tau, j + q)^{i+p} \quad i + m + 2q \leq n + j + 2p
\]

\[
(\Delta_1 \cup \Delta_2) \vdash x^{n+p} : \tau, m + q \tag{Var}
\]

Similar to the above sub-case.

Lemma 5 (Promotion). Let \(q \leq p\). If \(\Delta \vdash v^n : \tau, m\) then \(\Delta \cup (\Delta_2 \uparrow_{(q,p)}) \vdash v^n \uparrow^p : \tau, m + q\).

Proof. By induction on \(v^n\).

Lemma 6 (Substitution). If \(j < m\) and \(\Delta_1 \cup (x \mapsto (\tau', j)^i; \Delta_2) \vdash e^n : \tau, m\) and \(\Delta_1 \vdash v^i : \tau', j\) then one of the following three judgments holds.

1. \(\Delta_1 \vdash e^n[x^n := v^i \uparrow^{n-i}] : \tau, m\) if \(n > i\).
2. \(\Delta_1 \vdash e^n[x^n := v^i \downarrow^{n}] : \tau, m\) if \(n < i\)
3. \(\Delta_1 \vdash e^n[x^n := v^n] : \tau, m\), otherwise

Proof. By induction on the structure \(e^n\). If \(e^n = x^n\) then we have:

\[
\Delta(x) = (\tau, j)^i \quad m + i \leq n + j
\]

\[
(\Delta_1 \cup (x \mapsto (\tau, j)^i; \Delta_2) \vdash x^n : \tau, m)
\]

- If \(n < i\) and by the hypothesis \(j \leq m\) then \(m + i > n + j\). Hence \(\Delta_1 \cup (x \mapsto (\tau, j)^i; \Delta_2) \vdash x^n : \tau, m\) cannot be typable.
- If \(n > i\) then \(m - j < n - i\) and the Promotion Lemma 5 applies.
- \(i = n\) and by hypothesis \(j \leq m\) and \(m + i \leq n + j\) then \(j = m\). Then, \(\Delta_1 \vdash e^n[x^n := v^n] : \tau, m\).

Corollary 7 (\(\beta\) Rule). If \(\Delta \vdash ((\lambda x.e^n)^n v^n) : \tau, m\) then \(\Delta \vdash e^n[x^n := v^n] : \tau, m\).

Proof. By induction on the structure of \(e^n\). If \(e^n = x^n\) then we have:

\[
\Delta(x) = (\tau, j)^i \quad m + i \leq n + j
\]

\[
(\Delta_1 \cup (x \mapsto (\tau, j)^i; \Delta_2) \vdash x^n : \tau, m)
\]

Proposition 10. If \(\Delta \vdash e^n_1 : \tau, m\) and \(e^n_1 \rightarrow e^n_2\) then \(\Delta \vdash e^n_2 : \tau, m\).

Proof. By induction on the structure of \(e^n_1\). If the rewrite is at the root then use Lemmas 8 and 9, and Corollary 7. If \(e^n_1\) contains a redex then apply induction hypothesis.

Proposition 11 (Subject Reduction). If \(\Delta \vdash e^n_1 : \tau, m\) and \(e^n_1 \rightarrow e^n_2\) then \(\Sigma \vdash e^n_2 : \tau, m\).

Proof. By induction on the length of the derivation.
5.2 Faulty Terms

Lemma 12 (Faulty Terms are Not Typable). If $e \in F$ then there is no $\Delta, t, a$ such that
$\Delta \vdash e : t, a$.

Proof. By case analysis over the structure of $e$. Let $e = c_1[(\lambda x.c_2[x^n])^i]$ such that $n < i$, that is,
$i = n + k_i + 1$. Assume that $\Delta \vdash e : \tau, m$. This implies that $x \mapsto (\tau', j)^i \Delta' \vdash x^n : \tau', p$. This means
that $i + p \leq n + j$. Because $p = j + k_2$ then $j \leq p$. This implies that $n + k_1 + 1 + j + k_2 \leq n + j$
which is impossible. The other cases are straight-forward. \hfill \Box

6 Natural Semantics

In previous work, we defined core MetaML by a natural semantics [25]. While this style of presen-
tation is closer to the implementation of MetaML than the reduction semantics presented in this
paper, it is more complex. We have found that it was easier to prove type soundness first with
respect to the reduction semantics, and then to extend this result to the natural semantics.

In this paper, we present a more concise natural semantics for MetaML than the one we have
presented in previous work [25]:

\[
\frac{e_1 \equiv (\lambda x.e'0) \equiv \lambda x.e_0^0 \equiv v_1^0 \equiv (e_0^0 x : v_1^0)}{e_2 \equiv (\lambda x.e'0) \equiv \lambda x.e_0^0 \equiv v_2^0 \equiv (e_0^0 x : v_2^0)}
\]

A key property of this presentation is that it avoids the explicit use of a gensym or newname
function for renaming abstractions at levels greater than zero. This improvement avoids the prob-
lems that Moggi points out regarding the use of such stateful functions in defining the semantics
of two-level languages [18].

Now we move on to present some fundamental results about the untyped $\lambda$-R language, and
use these results, in addition to the soundness of the type system with respect to the reduction
semantics, to prove the soundness of the type system with respect to the natural semantics.

We say that two terms $e_1$ and $e_2$ are observationally equivalent, written $e_1 \sim e_2$, if for any
context $c[]$ such that both $c[e_1]$ and $c[e_2]$ are closed, then $c[e_1] \equiv v_1^0$ if and only if $c[e_2] \equiv v_2^0$,
and $v_1^0 = v_2^0$ if and only if $v_1^0 = v_2^0$ when both relations are defined.

Lemma 13. If $e^n \equiv v^n$ then $e^n \sim v^n$.

Proof. By induction on the proof tree for $e^n \equiv v^n$. \hfill \Box

Lemma 14. If $e \sim v$ then $e \equiv v$.

Proof. This proof requires a Standardization Theorem along the lines of Plotkin [20], but one
extended to deal with Brackets, Escape and Run. We omit the details for the sake of brevity.
Please see the technical report for the full details [24]. \hfill \Box
Corollary 15. There exists a value \( v \) such that \( \lambda \cdot R \vdash e = v \) if and only if \( e \leftrightarrow v' \).

Proof. Consequence of Lemmas 14 and 13. \qed

Theorem 16 (Soundness of Axiomatic Semantics). If \( \lambda \cdot R \vdash e_1 = e_2 \) then \( e_1 \sim e_2 \).

Proof. If \( e_1 \leftrightarrow v_1 \) then by Corollary 15 \( \lambda \cdot R \vdash e_1 = v_1 \). Hence, \( \lambda \cdot R \vdash e_2 = v_1 \). By Corollary 15, there exists a value \( v_2 \) such that \( e_2 \leftrightarrow v_2 \). By Lemma 13, \( \lambda \cdot R \vdash v_1 = v_2 \). Since the axiomatic semantics is Church-Rosser, we have \( v_1 \overset{\sim}{\rightarrow} v \) and \( v_2 \overset{\sim}{\rightarrow} v \). Thus, \( e_1 \sim e_2 \). \qed

We define undesirable behavior in the natural semantics in the classical manner: we introduce a new "value" Wrong, written \( T \), and a set of rules complementing the rules of the natural semantics, and returning \( T \) in all these new cases. We call the combination of these two sets of rules the augmented natural semantics, and denote it by \( \overset{T}{\sim} \).

Lemma 17. If \( e \overset{T}{\rightarrow} T \) then \( e \overset{T}{\rightarrow} f \) and \( f \in F \) and \( f \neq v \).

Proof. By induction on the proof tree of the augmented natural semantics \( \overset{T}{\rightarrow} \). \qed

Theorem 18 (Type Soundness). If \( \Delta \vdash e : \tau, m \) and \( e \overset{T}{\rightarrow} e' \) then \( e' \neq T \).

Proof. We prove the contrapositive. If \( e' = T \) and \( e \overset{T}{\rightarrow} T \) then by Lemma 17, \( e \overset{T}{\rightarrow} f \). Hence by type soundness of the reduction semantics, \( e \) is not typable. \qed

7 Related Work

Multi-stage programming techniques have been used in a wide variety of settings, including run-time program generation in ML [17], run-time specialization of C programs [5, 4, 21, 9], and advanced dynamic compilation for C programs [1].

Nielsen and Nielson present a seminal detailed study into a two-level functional programming language [19]. This language was developed for studying code generation. Davies and Pfenning show that a generalization of this language to a multi-level language called \( \lambda^\square \) gives rise to a type system very related to a modal logic, and that this type system is equivalent to the binding-time analysis of Nielsen and Nielson [7]. Intuitively, \( \lambda^\square \) provides a natural framework where LISP’s quote and eval can be present in a language. The semantics of our Bracket and Run correspond closely to those of quote and eval, respectively.

Glück and Jørgensen study partial evaluation in the generalized context where inputs can arrive at an arbitrary number of times rather than just specialization-time and run-time [12]. They also demonstrate that binding-time analysis in a multi-level setting can be done with efficiency comparable to that of two-level binding time analysis. Our notion of level is very similar to that used by Glück and Jørgensen[10, 11].

Davies extended the Curry-Howard isomorphism to a relation between modal logic and the type system for a multi-level language [6]. Intuitively, \( \lambda^\square \) provide a good framework for formalizing the presence of quote and quasi-quote in a language. The semantics of our Bracket and Escape correspond closely to those of quote and quasi-quote, respectively. Previous attempts to combine the \( \lambda^\square \) and \( \lambda^\square \) systems have not been successful [7, 6, 25]. To our knowledge, our work is the first successful attempt to define a sound type system combining Brackets, Escape and Run in the same language.

Moggi advocates a categorical approach to two-level languages, and and uses indexed categories to develop models for two languages similar to \( \lambda^\square \) and \( \lambda^\square \) [18]. He points out that two-level languages generally have not been presented along with an equational calculus. Our paper has eliminated this problem for MetaML, and to our knowledge, is the first presentation of a multi-level language using axiomatic and reductions semantics.
8 Conclusion

In this paper, we have presented an axiomatic and reduction semantics for a language with three staging constructs: Brackets, Escape, and Run. Arriving at the axiomatic and reduction semantics was of great value to enhancing our understanding of the language. In particular, it helped us to formalize an accurate syntactic characterization of faulty terms for this language. This characterization played a crucial role in leading us to the type system presented here. Finally, it is useful to note that our reduction semantics allows for $\beta$-reductions inside Brackets, thus giving us a basis for verifying the soundness of the safe-$\beta$ optimization that we discussed in previous work [25].

MetaML currently exists as a prototype implementation that we intend to distribute freely on the web. The implementation supports the three programming constructs, higher-order datatypes (with support for Monads), Hindley-Milner polymorphism, recursion, and mutable state. The system has been used for developing a number of small applications, including simply term-rewriting system, monadic staged compilers, and numerous small bench-mark functions.

We are currently investigating the incorporation of an explicit recursion operator and Hindley-Milner polymorphism into the type system presented in this paper.

Acknowledgements: We would like to thank John Matthews and Matt Saffell for comments on a draft of this paper.

References


The Anatomy of a Component Generator

Walid Taha & Jim Hook
(walidt, hook)@cse.ogi.edu
The Oregon Graduate Institute

In this extended abstract, we outline some essential elements of a conceptual model for a component generation system. This model is based on an extensive study of a large number of high-level program generation systems, and the significant body of related literature. We focus our attention on the architectural elements of this model, and briefly discuss the technological and process elements. We show how the model is a useful basis for comparing component generation technologies. With a rapidly growing area like component generation, it is hard to get a truly representative sample of generators. As a workaround, we illustrate our model using seven significant component generation systems developed by various research groups, and discuss some insights that the model provides. We conclude with an overview of the current status of our investigation.

1. The Pragmatic Need for Models

We know that component generation can be very beneficial for evolving systems, but we don't have a widely-accepted conceptual model for component generation systems. Conceptual models allow us to categorize and distill our knowledge of details into more manageable and structured information. We believe that such a model would facilitate better communication of ideas, within our own research group (PacSoft), within the component generation research area, within the programming languages area, and with the outside world. For example, it will necessarily play an important role in transferring our ideas as a research community to software houses that can develop industry-strength, general purpose component generators.

We have been working towards such a model for almost three years now, and have studied over 100 related publications, in addition to being involved in PacSoft's SDRR component generation project [KMB96]. Why has it taken so much effort? The major hurdle is that interesting component generation systems emerge from many corners of computer science, which often means incompatible vocabularies. For example, the word "Component" can have significantly different meanings in different papers. The diversity of programming languages, operating systems, and tools used in developing the generators, and of the researchers' expectations from all of these, add significantly to the difficulty of understanding the literature in a manner that would allows us to compare and contrast two different generation technologies.

2. The Architectural Element

Software architectures [PW92] communicate ideas about software systems, and are especially useful when parties involved come from a variety of different backgrounds. Architectural descriptions provide an abstract basis for our model, a basis that is independent of the technology underlying the generator, the development process, and the application domain.

Even when composed of relatively simple subsystems, the collective architecture of a generator is often quite complex, and involves a significant number of distinct artifacts and users. Artifacts include the generator, the input and output of the generator, libraries, and the legacy system hosting the generated component. Users include the developers of the generator, it's input, and the libraries. Ideally, the input to the generator is a simple, compact specification that is easy to maintain. However, it is often the case that an executable program cannot be generated solely from such

---

1 This research is supported by a contract with the USAF Materiel Command. Contract F19628-93-C-0069.
2 In this paper, it will mean CORBA/COM-like components.
specifications. Therefore, it is common to find an additional (specification) language, often in the form of annotations, for controlling the generator. There may even be a developer dedicated to this task.

Hence, a model for component generators should admit all possible answers to the following questions:

- What is the input to the generator? Who writes this input?
- What is the output of the generator? Who uses it?
- What libraries does the output use? Who writes these libraries?
- How does the generator work? Who wrote it, and how? How is it controlled?
- With what systems does the generated component interact?

While it is not common to consider all of these dimensions of variability simultaneously, this is precisely what is needed when we wish to relate and contrast more than one existing component generation system. The figure below is a schematic\(^1\) representing the minimal architectural schema that arises if the answer to the each of the above questions is distinct.

\[\text{Diagram of component generator architecture}\]

The figure above explicates the implicit complexity of even the simplest generative system. For instance, consider the yacc parser-generator [Joh75]. Development work on the generator itself has stopped, and hence, we usually don't think of either the developer or the source yacc.c. The generator input is the grammar proper, and the control annotations are the directives regarding precedence and association. Note that control annotations need not be in a separate file. The component developer and the generator controller are the same person. The grammar file could also contain further control instructions about what library files the generator output might be using. The libraries used by the generator output include lib.y.c, which contains the abstract machine for the parse table. The interface is usually header files describing the legacy system functions that the parser uses. Finally, while we rarely see a user directly interacting\(^2\) with the parser generated by yacc, the user of the legacy system is, indirectly, the component user.

\(^1\) Drawn in the Generator Description Language, GDL [TS97].

\(^2\) Interaction commutes, and hence, we could have drawn the component user directly connected to the generated component, and the diagram would have had the same meaning.
2.1 Basic Distinguishing Characteristics

Certain aspects of the architecture sketched in the last section are "not negotiable": a generative architecture has to include a generator, a generator input, and a generated component. And every artifact that is not mechanically generated must have an author. The architecture described above gives us a very natural basis for our model that captures these essential invariants. However, it offers too many dimensions of variability. The design space is indeed vast. But some of these dimensions are more informative than others, in that they are better discriminators between various component generation systems. We have identified basic distinguishing characteristics:

1. Who is the primary user, that is, the "customer" the system is intended to benefit?
2. What expertise is expected from the main user?
3. Which users are distinct, and which users are not? For example, is the role of generator development identified with the role of generator control?
4. Does the generator have a distinct notion of control annotations?

These factors are derived or computed from the architectural variabilities. In the following section, we illustrate the relevance of these criteria by considering some important generative systems.

2.2 Application to Seven Research Component Generation Systems

For brevity, we will not review all the systems we have studied. Instead, we present summary of our observations, and then illustrate how these observation can be interpreted. In the following table, "=" between two different kinds of users means that we did not find them to be treated differently. In cases where there is no explicit notion of control annotations, the input to the generator can be viewed as being an "Implicit" control specification:

<table>
<thead>
<tr>
<th>Systems</th>
<th>Primary User(s)</th>
<th>Primary User's Expertise</th>
<th>Distinct Users</th>
<th>Control Annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIP [MKS97]</td>
<td>CU</td>
<td>Domain expert</td>
<td>CU=CD=GC, GD, LD</td>
<td>Implicit</td>
</tr>
<tr>
<td>GenVoca [BST+94]</td>
<td>LD</td>
<td>Programmer</td>
<td>CU, CD=GC=LD, GD</td>
<td>Design rules</td>
</tr>
<tr>
<td>KIDS / SpecWare [Smi90, SJ94]</td>
<td>CD</td>
<td>Formal methods expert</td>
<td>CU, CD=GC=LD, GD</td>
<td>Refinements</td>
</tr>
<tr>
<td>SDRR [BH+94, KMB96]</td>
<td>GD, CD</td>
<td>Domain expert</td>
<td>CU=CD, GC=GD=LD</td>
<td>Implicit</td>
</tr>
<tr>
<td>Amphion [LPP+94]</td>
<td>CU</td>
<td>Domain expert</td>
<td>CU=CD, GC=GD, LD</td>
<td>Implicit</td>
</tr>
<tr>
<td>AOP [GLM+97]</td>
<td>CD</td>
<td>Programmer</td>
<td>CU, CD=GC=GD=LD</td>
<td>Aspects</td>
</tr>
</tbody>
</table>

Let us consider the first case: In the ISI technology, the generator developer (GD) uses the POPART metaprogramming tool-kit and a relational extension of C or Java to develop the generator [Bal92,Wil81,Wil90]. In the literature we surveyed, the roles of the generator controller (GC) and generator developer were not distinguishable. Pragmas are used to guide the relational compiler as to how to implement relations.
The following sub-sections discuss two of the main observations that can be drawn on the basis of this information.

2.2.1 What to Mix, and What to Match

Consider the kind of information that might interest a software engineer interested in building a component generator. Some technologies address similar classes of users, such as ISI and SDRR, and MIP and Amphion. This means that these technologies could be a good basis for synthetic systems combining the benefits of both. For example, SDRR’s technology, which leverages on functional programming, can benefit greatly from ISI’s meta-programming technology, and vise versa. When a basic distinguishing characteristic identifies two systems, there are usually many other (often less-abstract) dimensions in which they are different. For example, MIP and Amphion fall on distinct points along the dimension of real-time constraints. We consider this dimension to be somewhat less abstract than architecture because it is more dependent on the application domain. Some of these dimensions should be in a model for component generators, discussed in the next section.

Other technologies address users that are usually not emphasized by others. For example, GenVoca is unique in addressing concerns of the library developer (LD). This suggests that high-level ideas from the GenVoca system might be readily combinable with generation technologies covered in our survey.

2.2.2 How to Control Generation

Four very different kinds of annotations are being considered by three different groups, namely, ISI’s pragmas, GenVoca’s design-rules, KIDS and SpecWare refinements, and AOP’s aspects. These annotations are an important characteristic of modern component generation systems that was not commonplace in earlier transformational programming systems.

Control annotations can be viewed as Domain-Specific Languages (DSLs). For example, yacc’s specifications for precedence of operators is one such DSL. In this light, we can say that the first three kinds of annotations are single languages, and AOP’s aspects can be thought of as families of DSLs. We believe that the study of these generator-control DSLs will play an important role in developing general-purpose, industry-standard component generation systems.

3. Technology and Process Elements

Our model also includes two other elements; the technology underlying the generation system, and the process by which the generator itself is developed. Both can be viewed as refinements of the architectural model. The following table summarizes some distinguishing characteristics of the systems surveyed:

<table>
<thead>
<tr>
<th>Systems</th>
<th>Underlying Technology</th>
<th>Generator Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISI</td>
<td>Meta-programming calculus and tools</td>
<td>Using POPART tools and relational C or Ada</td>
</tr>
<tr>
<td>MIP</td>
<td>Model-integrated real-time control</td>
<td>Using the MIP paradigm</td>
</tr>
<tr>
<td>GenVoca</td>
<td>Algorithm selection and object-orientation</td>
<td>Using design rules to specify acceptable library combinations</td>
</tr>
<tr>
<td>KIDS / SpecWare</td>
<td>Formal verification</td>
<td>Using specifications and refinements to characterize and derive programs</td>
</tr>
<tr>
<td>SDRR</td>
<td>Typed, functional programming</td>
<td>Using SDRR to create the front-end of the SDRR pipeline</td>
</tr>
</tbody>
</table>
Amphion | Theorem proving and program synthesis | Using Meta-Amphion, a theory of the domain, and an inference engine
---|---|---
AOP | AOP | Using (any technology?) to develop a weaver and aspects

4. Conclusion

We have outlined a model for component generation systems that we are currently developing. The model captures some of the bare essentials required for an object of study to be considered a generator, without going too deeply into the details of any particular system. We illustrated how it admits simple, clear, and objective criteria for comparing component generation systems. Our work shows that there is significant diversity not only in the cultures and application domains of contemporary component generation research projects, but also in technical problems that are unique to the emerging research area of component generation, such widespread interest in generation control.

Acknowledgments: We thank Lisa Walton, Andrew Black, Sherri Shulman, Tito Autry, Therese Fisher, Shailish Godbole, and Amol Vyas for valuable discussions.

References

Optimizing ML Using a Hierarchy of Monadic Types

Andrew Tolmach*
Pacific Software Research Center
Portland State University & Oregon Graduate Institute
apt@cs.pdx.edu
March 9, 1998

Abstract

We describe a type system and typed semantics for call-by-value functional languages that use a hierarchy of monads to describe and delimit a variety of effects, including non-termination, exceptions, and state. The type system and semantics can be used to organize and justify a variety of optimizing transformations in the presence of effects. In addition, we describe a simple monad inferencing algorithm that computes the minimum effect for each subexpression of a program, and provides more accurate effects information than local syntactic methods.

*Supported, in part, by the US Air Force Materiel Command under contract F19628-93-C-0069 and by the National Science Foundation under grant CCR-9503383.
1 Introduction

Optimizers are often implemented as engines that repeatedly apply improving transformations to programs. Among the most important transformations are propagation of values from their defining site to their use site, and hoisting of invariant computations out of loops. If we use a pure (side-effect-free) language based on the lambda calculus as our compiler intermediate language, these transformations can be neatly described by the simple rules for beta-reduction

\[(\text{Beta}) \quad \text{Let } x = e \text{ in } b = b[e/x]\]

and for the interchange and lifting of bindings

\[(\text{Exchange}) \quad \text{Let } x_1 = e_1 \text{ in } (\text{Let } x_2 = e_2 \text{ in } b) = \text{Let } x_2 = e_2 \text{ in } (\text{Let } x_1 = e_1 \text{ in } b)\]

\[\quad (x_1 \notin FV(e_2); x_2 \notin FV(e_1))\]

\[(\text{RecHoist}) \quad \text{Letrec } f = (\lambda x. \text{let } y = e_1 \text{ in } e_2) \text{ in } b = \text{let } y = e_1 \text{ in } (\text{letrec } f = \lambda x. e_2 \text{ in } b)\]

\[\quad (x, f \notin FV(e_1); y \notin FV(b))\]

where the side conditions nicely express the data dependence conditions under which the transformations are valid.\(^1\) Effective compilers for pure, lazy functional languages (e.g., [10]) have been conceived and built on the basis of such transformations, with considerable advantages for modularity and correctness.

It would be nice to apply similar methods to the optimization of languages like ML, which have side effects such as I/O, mutable state, and exceptions. Unfortunately, these “rearranging” transformations are not generally valid for such languages. For example, if we apply (Beta) in a situation where evaluating \(e\) performs output and \(x\) is mentioned twice in \(b\), evaluating the resulting expression might produce the output twice. In fact, once an eager evaluation order is fixed, even non-termination becomes a “side effect.” For example, (RecHoist) is not valid unless \(e_1\) is known to be terminating (and free of other effects too, of course).

A similar challenge long faced lazy functional languages at the source level: how can we give the power of side-effecting operations without invalidating simple “equational reasoning” based on (Beta) and similar rules? The effective solution discovered in that context is to use monads [8, 12]. An obvious idea, therefore, is to use monads in an internal representation (IR) for compilers of call-by-value languages. Some initial steps in this direction were recently taken by Peyton Jones, Launchbury, Shields, and Tolmach [11]. The aim of that work was to design an IR suitable for both eager and lazy source languages. In this paper we pursue the use of monads with particular reference to eager languages (only), and address the question of how to discover and record several different sorts of effects in a single, unified monadic type system. We introduce a hierarchy of

\(^1\)Of course, the fact that a transformation is valid doesn’t mean that applying it will necessarily improve the program. For example, (Beta) is not an improving transformation if \(e\) is expensive to compute and \(x\) appears many times in \(b\); similarly, (RecHoist) is not improving if \(f\) is not applied in \(b\).
monads, ordered by increasing "strength of effect," and an inference algorithm for annotating
source program subexpressions with their minimal effect.

Past approaches to coping with effects have fallen into two main camps. One approach approach
(used, e.g., by SML of New Jersey [2] and the TIL compiler [16]) is to fall back on a weaker form
of (Beta), called (BetaJ, which is valid in eager settings. (BetaV) restricts the bound expression
e to variables, constants, and λ-abstractions; since "evaluating" these expressions never actually
causes any computation, they can be moved and substituted with impunity. To augment this rule,
these compilers use local syntactic analysis to discover expressions that are demonstrably pure and
terminating. These analyses cannot "see through" function calls, but they can be quite effective,
particularly if the compiler inlines functions enthusiastically. The other approach (used, e.g., by
the ML Kit compiler [4]) uses a sophisticated effect inference system [14] to track the latent effects
of functions on a very detailed basis. The goals of this school are typically more far-reaching; the
aim is to use effects information to provide more generous polymorphic generalization rules (e.g.,
as in [19, 15]), or to perform significantly more sophisticated optimizations, such as automatic
parallelization or stack-allocation of heap-like data. In support of these goals, effect inference has
generally been used to track store effects at a fine-grained level.

Our approach is essentially a simple monomorphic variant of effect inference applied to a wider
variety of effects (including non-termination, exceptions, and IO), cast in monadic form, and in-
tended to support transformational code-motion optimizations. We infer information about latent
effects, but we do not attempt to calculate effects at a very fine level of granularity. In return,
our inference system is particularly simple to state and implement. However, there is nothing
fundamentally new about our system as compared with that of Talpin and Jouvelot [14], except
our decision to use a monadic syntax and validate it using a typed monadic semantics. A practical
advantage of the monadic syntax is that it makes it easy to reflect the results of the effect inference
in the program itself, where they can be easily consulted (and kept up to date) by subsequent
optimizations, rather than in an auxiliary data structure. An advantage of the monadic semantics
is that it provides a natural foundation for probing and proving the correctness of transformations
in the presence of a variety of effects.

In related work, Wadler [18] has recently and independently shown that Talpin and Jouvelot's
effect inference system can be applied in a monadic framework; he uses an untyped semantics, and
considers only store effects. In another independent project, Benton and Kennedy are prototyping
an ML compiler using a monadic encoding similar to ours [3].

2 Source Language

This section briefly describes an ML-like source language we use to explain our approach. The
call-by-value source language is presented in Figure 1. It is a simple, monomorphic variant of ML,
expressed in A-normal form [5], which explicitly binds a name to the result of each computation
and makes evaluation order completely explicit. The class const includes primitive functions as
well as constants. The Let construct is monomorphic; that is, \texttt{Let}(x,e,b) has the same semantics and typing properties as would \texttt{App}(\texttt{Abs}(x,b),e) (were this legal A-normal form). The restriction to a monomorphic language is not essential; see Section 5. All functions are unary; primitives like \texttt{Plus} take a two-element tuple as argument. For simplicity of presentation, we restrict \texttt{Letrec} to single functions.

The language is not explicitly typed, but the underlying types include the base types Int, Bool, and Exn, tuples, and arrows. We use tuples as a surrogate for more general algebraic datatypes; to permit type inference for Projects in the absence of declarations, we provide the total size of the tuple as an additional parameter. We assume a supply of appropriate constants for each base type. Exceptions carry values of type Exn, which are nullary exception constructors. \texttt{Raise} takes an exception constructor; rather than providing a means for declaring such constructors, we assume an arbitrary pool of constructor constants. \texttt{Handle} catches all exceptions that are raised while evaluating its first argument and passes the associated exception value to its second argument, which must be a handler function expecting an Exn. The body of the handler function may or may not choose to reraise the exception depending on its value, which may be tested using \texttt{EqExn}. The primitive function \texttt{Divide} has the potential to raise a particular exception \texttt{DivByZero}. We supply \texttt{WriteInt} as a paradigmatic state-altering primitive; internal side-effects such as ML reference manipulations would be handled similarly. All other primitives are pure and guaranteed to terminate. The semantics of the remainder of the language are completely ordinary.

3 Intermediate Language with Monadic Types

Figure 2 shows the abstract syntax of our monadic intermediate representation (IR). (For an example of the code, look ahead to Figure 10.) For the most part, terms are the same as in the source language, but with the addition of monad annotations on Let and Handle constructs and a new Up construct; these are described in detail below. In addition, identifiers (and \texttt{Raise} expressions)
Figure 2: Abstract Syntax for Monadic Typed Intermediate Language.

are explicitly typed, in order that we may easily compute the type of any closed expression.

Values have ordinary value types (\(v\text{types}\)); expressions have monadic types (\(m\text{types}\)), which incorporate a \(v\text{type}\) and a monad (possibly the ID monad). Since this is a call-by-value language, the domain of each arrow types is a \(v\text{type}\), but the codomain is an arbitrary \(m\text{type}\). The typing rules are given in Figure 3. In this figure, and throughout our discussion, \(t\) ranges value types, \(m\) over monads, \(v\) over values, \(c\) over constants, \(x\) over variables, and \(e\) over expressions. The initial type environment is described in Figure 4.

For this presentation, we use four monads arranged in a simple linear order. In order of “increasing effect” these are:

- **ID**, the identity monad, which describes pure, terminating computations.
- **LIFT**, the lifting monad, which describes pure but potentially non-terminating computations.
- **EXN**, the monad of exceptions and lifting, which describes computations that may raise an (uncaught) exception, and are potentially non-terminating.
- **ST**, the monad of state, exceptions, and lifting, which describes computations that may write to the “outside world,” may raise an exception, and are potentially non-terminating.

We write \(m_1 < m_2\) iff \(m_1\) precedes \(m_2\) on this list. Intuitively, \(m_1 < m_2\) implies that computations in \(m_2\) are “more effectful” than those in \(m_1\); they can provoke any of the effects in \(m_1\) and then some. This particular hierarchy captures most of the interesting distinctions and still gives us a simple inference algorithm (see Section 5). More elaborately stratified monadic structure is certainly possible; we discuss this in more detail below.

More formally, \(m_1 < m_2\) implies that there exists an embedding \(wp_{m_1 \rightarrow m_2}\) which, for every value type \(t\), maps the domain corresponding to \(M(m_1, t)\) into the domain corresponding to \(M(m_2, t)\). The
\[
E(v) = t \\
\frac{\quad E \vdash_v \text{Var} \ v : t}{E}
\]

\[
\text{Typeof}(c) = t \\
\frac{\quad E \vdash_v \text{Const} \ c : t}{E}
\]

\[
E \vdash_v v : t \\
\frac{\quad E \vdash_v \text{Val} \ v : \text{M}(\text{ID}, t)}{E}
\]

\[
E \vdash \{x : t_1\} \vdash e : \text{M}(m_2, t_2) \\
\frac{\quad E \vdash \text{Abs} \ (x : t_1, e) : \text{M}(\text{ID}, t_1 \rightarrow \text{M}(m_2, t_2))}{E}
\]

\[
E \vdash_v v_1 : t_1 \rightarrow \text{M}(m_2, t_2) \\
E \vdash_v v_2 : t_1 \\
\frac{\quad E \vdash \text{App} \ (v_1, v_2) : \text{M}(m_2, t_2)}{E}
\]

\[
E \vdash_v v : \text{Bool} \\
E \vdash e_1 : \text{M}(m, t) \\
E \vdash e_2 : \text{M}(m, t) \\
\frac{\quad E \vdash \text{If} \ (v, e_1, e_2) : \text{M}(m, t)}{E}
\]

\[
E \vdash e_1 : \text{M}(m_1, t_1) \\
E \vdash \{x : t_1\} \vdash e_2 : \text{M}(m_2, t_2) \\
(m_1 \leq m_2) \\
\frac{\quad E \vdash \text{Let} \ (m_1, m_2, x : t_1, e_1, e_2) : \text{M}(m_2, t_2)}{E}
\]

\[
E \vdash \{f : t_0 \rightarrow \text{M}(m_1, t_1), x : t_0\} \vdash e_1 : \text{M}(m_1, t_1) \\
E \vdash \{f : t_0 \rightarrow \text{M}(m_2, t_1)\} \vdash e_2 : \text{M}(m_2, t_2) \\
\text{(LIFT} \leq m_1) \\
\frac{\quad E \vdash \text{Letrec} \ (f : t_0 \rightarrow \text{M}(m_1, t_1), x : t_0, e_1, e_2) : \text{M}(m_2, t_2)}{E}
\]

\[
E \vdash_v v_1 : t_1 \ldots \\
E \vdash_v v_n : t_n \\
\frac{\quad E \vdash \text{Tuple} \ (v_1, \ldots, v_n) : \text{M}(\text{ID}, \text{Tup}(t_1, \ldots, t_n))}{E}
\]

\[
E \vdash_v v : \text{Tup}(t_1, \ldots, t_n) \\
(0 \leq i < n) \\
\frac{\quad E \vdash \text{Project} \ (i, n, v) : \text{M}(\text{ID}, t_i)}{E}
\]

\[
E \vdash_v v : \text{Exn} \\
\frac{\quad E \vdash \text{Raise} \ (\text{M}(\text{EXN}, t), v) : \text{M}(\text{EXN}, t)}{E}
\]

\[
E \vdash e : \text{M}(m, t) \\
E \vdash_v v : \text{Exn} \rightarrow \text{M}(m, t) \\
\text{(EXN} \leq m) \\
\frac{\quad E \vdash \text{Handle} \ (m, e, v) : \text{M}(m, t)}{E}
\]

\[
E \vdash e : \text{M}(m_1, t) \\
(m_1 \leq m_2) \\
\frac{\quad E \vdash \text{Up} \ (m_1, m_2, e) : \text{M}(m_2, t)}{E}
\]

Figure 3: Typing rules for intermediate language
Figure 4: Typings for constants in initial environment

$\text{Integer} \ = \ \text{Int} \\
\text{True, False} \ = \ \text{Bool} \\
\text{DivByZero} \ = \ \text{Exn} \\
\text{Plus, Minus, Times} \ = \ \text{Arrow(Tup[Int, Int], M(ID, Int))} \\
\text{Divide} \ = \ \text{Arrow(Tup[Int, Int], M(EXN, Int))} \\
\text{EqInt, LtInt} \ = \ \text{Arrow(Tup[Int, Int], M(ID, Bool))} \\
\text{EqBool} \ = \ \text{Arrow(Tup[Bool, Bool], M(ID, Bool))} \\
\text{EqExn} \ = \ \text{Arrow(Tup[Exn, Exn], M(ID, Bool))} \\
\text{WriteInt} \ = \ \text{Arrow(Int, M(ST, Tup[]))}$

$\text{wp}_{\text{ID} \rightarrow m}(e)$ is equivalent to $\text{unit}_m(e)$. Each monad $m$ also has a conventional $\text{bind}_m$ operation which serves to compose computations in $m$. Figure 5 gives semantic interpretations for types as complete partial orders (CPO’s), and for our monads, together with the associated $\text{up}$ and $\text{bind}$ functions. Note that the $\text{up}$ functions are defined in such a way that they compose, i.e., for all $m_0 \leq m_1 \leq m_2$, we have $\text{up}_{m_0 \rightarrow m_2} = \text{up}_{m_1 \rightarrow m_2} \circ \text{up}_{m_0 \rightarrow m_1}$.

A typed semantics for terms is given in Figures 6 and 7. Environments $\rho$ map identifiers to values. This semantics is largely straightforward. However, the Let construct now serves to make the composition of monadic computations explicit, and the Up construct makes monadic coercions explicit. Intuitively,

$$\text{Let}(m_1, m_2, (x, t), e_1, e_2)$$

evaluates $e_1$, which has monadic type $\text{M}(m_1, t)$, performing any associated effects, binds the resulting value to $x : t_1$, and then evaluates $e_2$, which has monadic type $\text{M}(m_2, t_2)$. Thus, it essentially plays the role of the usual monadic $\text{bind}$ operation; in particular, if $m_1 = m_2$, the semantic interpretation of the above expression in environment $\rho$ is just

$$\text{bind}_{m_1}(\text{E}[e_1]\rho)(\lambda y.\text{E}[e_2]\rho[x := y])$$

However, our typing rules (Figure 3) require only that $m_2 \geq m_1$; i.e., $e_2$ may be in a more effectful monad than $e_1$. The semantics of a general “mixed-monad” Let is

$$\text{bind}_{m_2}(\text{up}_{m_1 \rightarrow m_2}(\text{E}[e_1]\rho))(\lambda y.\text{E}[e_2]\rho[x := y])$$

The term $\text{Let(Up}(m_1, m_2, e_1), (x, t), e_1, e_2)$ has the same semantics, so the more general form of Let is strictly redundant. But this form is useful, because it makes it easier to state (and recognize left-hand sides for) many interesting transformations involving Let whose validity depends on the monad $m_1$ rather than on $m_2$. For example, a “non-monadic” Let, for which (Beta) is always valid, is simply one in which $m_1 = \text{ID}$. Further examples will be shown in the next section.

The semantics of the “non-proper morphism” $\text{Handle}(e, v)$ deserve special attention. Expression $e$ may be in either EXN or ST, and the meaning of $\text{Handle}$ depends on which; the ST version
\[ T : \mathit{vtyp} \rightarrow \mathbb{CPO} \]
\[ T[\text{Int}] = \mathbb{Z} \]
\[ T[\text{Bool}] = \mathbb{Z} \]
\[ T[\text{Exn}] = \mathbb{Z} \quad (0 \text{ represents false}) \]
\[ T[\text{Tup}(t_1, \ldots, t_n)] = T[t_1] \times \ldots \times T[t_n] \quad (n > 0) \]
\[ T[\text{Arrow}(t_1, M(m_2, t_2))] = T[t_1] \rightarrow M[m_2](T[t_2]) \]

\[ M : \text{monad} \rightarrow \mathbb{CPO} \rightarrow \mathbb{CPO} \]
\[ M[\text{ID}]c = c \]
\[ M[\text{LIFT}]c = c_\bot \]
\[ M[\text{EXN}]c = (\text{Ok}(c) + \text{Fail}(\bot))_\bot \]
\[ M[\text{ST}]c = \text{State} \rightarrow ((\text{Ok}(c) + \text{Fail}(\bot)) \times \text{State})_\bot \]

\[ \begin{align*}
\text{bind}_{\text{ID}} x k &= k x \\
\text{bind}_{\text{LIFT}} x k &= k a \\
& \quad \text{if } x = a_\bot \\
& \quad \bot \quad \text{if } x = \bot \\
\text{bind}_{\text{EXN}} x k &= k a \\
& \quad \text{if } x = \text{Ok}(a)_\bot \\
& \quad \text{Fail}(b)_\bot \quad \text{if } x = \text{Fail}(b)_\bot \\
& \quad \bot \quad \text{if } x = \bot \\
\text{bind}_{\text{ST}} x k s &= k a s' \\
& \quad \text{if } s = (\text{Ok}(a), s')_\bot \\
& \quad (\text{Fail}(b), s')_\bot \quad \text{if } s = (\text{Fail}(b), s')_\bot \\
& \quad \bot \quad \text{if } x s = \bot \\
\end{align*} \]

\[ \begin{align*}
\text{up}_{m \rightarrow m} x &= x \\
\text{up}_{\text{ID} \rightarrow \text{LIFT}} x &= x_\bot \\
\text{up}_{\text{ID} \rightarrow \text{EXN}} x &= \text{Ok}(x)_\bot \\
\text{up}_{\text{ID} \rightarrow \text{ST}} x s &= (\text{Ok}(x), s)_\bot \\
\text{up}_{\text{LIFT} \rightarrow \text{EXN}} x &= \text{Ok}(a)_\bot \\
& \quad \bot \quad \text{if } x = a_\bot \\
\text{up}_{\text{LIFT} \rightarrow \text{ST}} x s &= (\text{Ok}(a), s)_\bot \\
& \quad \bot \quad \text{if } x = a_\bot \\
\text{up}_{\text{EXN} \rightarrow \text{ST}} x s &= (\text{Ok}(a), s)_\bot \\
& \quad (\text{Fail}(b), s)_\bot \quad \text{if } x = \text{Ok}(a)_\bot \\
& \quad \bot \quad \text{if } x = \text{Fail}(b)_\bot \\
\end{align*} \]

Figure 5: Semantics of Types and Monads
\[ \mathcal{V} : \text{value : } t \rightarrow \text{Env } \rightarrow \mathcal{T}[t] \]
\[ \mathcal{V}[\text{Var } v]_{\rho} = \rho(v) \]
\[ \mathcal{V}[\text{Const (Integer } i)]_{\rho} = i \]
\[ \mathcal{V}[\text{Const True}]_{\rho} = 1 \]
\[ \mathcal{V}[\text{Const False}]_{\rho} = 0 \]
\[ \mathcal{V}[\text{Const Plus}]_{\rho} = \text{plus} \]
\[ \mathcal{V}[\text{Const Divide}]_{\rho} = \text{divideby} \]
\[ \mathcal{V}[\text{Const WriteInt}]_{\rho} = \text{writeint} \]
\[ \mathcal{V}[\text{Const DivByZero}]_{\rho} = \text{divby0} \]

\[ \ldots \]
\[ \text{plus } (a_1, a_2) = a_1 + a_2 \]
\[ \text{divideby } (a_1, a_2) = \begin{cases} \text{Ok}(a_1/a_2)_\bot & \text{if } a_2 \neq 0 \\ \text{Fail}(\text{divby0})_\bot & \text{if } a_2 = 0 \end{cases} \]
\[ \text{State} = [Z] \] (sequence of integers written so far)
\[ \text{writeint } a \ s = (\text{Ok}(), \text{append}(s, [a]))_\bot \]
\[ \text{divby0} = 42 \]

Figure 6: Semantics of Values

must manipulate the state component. Note that there are two plausible ways to combine state with exceptions; in our semantics we have given (as in ML), the state is not reverted when an exception is handled. Incidentally, we don’t have to give a semantics when \( c \) is in ID or LIFT, because the typing rule for Handle disallows these cases. Of course, these cases might appear in source code; when typed IR is generated for them, \( c \) must be coerced into EXN with an explicit Up.\(^2\) A Raise expression is handled similarly; the typing rules force it into monad EXN, so semantics need only be given for that case, but the whole expression may be coerced into ST by an explicit Up if necessary.

As mentioned above, our basic approach is not restricted to the totally-ordered set of monads presented here. It extends naturally to any collection of monads forming a finite upper semi-lattice under the up embedding operation. It does not suffice to have a partial order; we insist that any two monads in the collection have a least upper bound with respect to embedding, so that we can always find a unique monad into which two arbitrary expressions (e.g., the two arms of an if) can be coerced. One might be tempted to describe such a lattice by specifying a set of “primitive” monads encapsulating individual effects, and then assuming the existence of arbitrary “union” monads representing combinations of effects. As the Handle discussion indicates, however, there is often more than one way to combine two effects, so that it makes no sense to talk in a general way about the “union” of two monads. Instead, it appears necessary to specify explicitly, for every

\(^2\)Another possibility is to drop the entire Handle in favor of \( c \), which by its type cannot raise an exception!
\[
\begin{align*}
\mathcal{E} : (\text{exp} : M(m,t)) \rightarrow E_{m} &\rightarrow M[m]\{T[t]\} \\
\mathcal{E}[\text{Val } v]_{\rho} &= \mathcal{V}[v]_{\rho} \\
\mathcal{E}[\text{Abs}(x,e)]_{\rho} &= \lambda y.\mathcal{E}[e]_{\rho}[x := y] \\
\mathcal{E}[\text{App}(v_{1},v_{2})]_{\rho} &= (\mathcal{V}[v_{1}]_{\rho}) (\mathcal{V}[v_{2}]_{\rho}) \\
\mathcal{E}[\text{If}(v,e_{1},e_{2})]_{\rho} &= \text{if } (\mathcal{V}[v]_{\rho}) (\mathcal{E}[e_{1}]_{\rho}) (\mathcal{E}[e_{2}]_{\rho}) \\
\mathcal{E}[\text{Let}(f,x,e_{1},e_{2})]_{\rho} &= \mathcal{E}[e_{2}]_{\rho}[f := f' (\lambda f'.\lambda v.\mathcal{E}[e_{1}]_{\rho}[f' := f', x := v])] \\
\mathcal{E}[\text{Tuple}(v_{1},\ldots,v_{n})]_{\rho} &= (\mathcal{V}[v_{1}]_{\rho},\ldots,\mathcal{V}[v_{n}]_{\rho}) \\
\mathcal{E}[\text{Project}(i,n,v)]_{\rho} &= \text{proj}_{i}(\mathcal{V}[v]_{\rho}) \\
\mathcal{E}[\text{Raise}(M(\text{EXN},t),v)]_{\rho} &= (\text{Fail}(\mathcal{V}[v]_{\rho}))_{\bot} \\
\mathcal{E}[\text{Handle}(m,e,v)]_{\rho} &= \text{handle}_{m}(\mathcal{E}[e]_{\rho})(\mathcal{V}[v]_{\rho}) \\
\mathcal{E}[\text{Let}(m_{1},m_{2},x,e_{1},e_{2})]_{\rho} &= \text{bind}_{m_{2}}(\text{up}_{m_{1} \rightarrow m_{2}}(\mathcal{E}[e_{1}]_{\rho}))(\lambda y.\mathcal{E}[e_{2}]_{\rho}[x := y]) \\
\mathcal{E}[\text{Up}(m_{1},m_{2},e)]_{\rho} &= \text{up}_{m_{1} \rightarrow m_{2}}(\mathcal{E}[e]_{\rho}) \\
\text{if } v a_{i} a_{f} &= a_{i} \quad \text{if } v \neq 0 \\
&= a_{f} \quad \text{if } v = 0 \\
\text{proj}_{i}(v_{1},\ldots,v_{n}) &= v_{i} \\
\text{handle}_{\text{EXN}} x h &= \text{Ok}(a)_{\bot} \quad \text{if } x = \text{Ok}(a)_{\bot} \\
&= h a \quad \text{if } x = \text{Fail}(a)_{\bot} \\
&= \bot \quad \text{if } x = \bot \\
\text{handle}_{\text{ST}} x h s &= (\text{Ok}(a),s')_{\bot} \quad \text{if } x s = (\text{Ok}(a),s')_{\bot} \\
&= h a s' \quad \text{if } x s = (\text{Fail}(a),s')_{\bot} \\
&= \bot \quad \text{if } x s = \bot 
\end{align*}
\]
(IdentUp) \[ \text{Up}(m, m, e) = e \]

(ComposeUp) \[ \text{Up}(m_0, m, e) = \text{Up}(m_1, m_2, (\text{Up}(m_0, m_1, e))) \quad (m_0 \leq m_1 \leq m_2) \]

(MonadId) \[ \text{Let}(m_2, m_3, x, \text{Up}(m_1, m_2, e), b) = \text{Let}(m_1, m_3, x, e, b) \]

(MonadId2) \[ \text{Let}(m_1, m_2, x, e, \text{Up}(\text{ID}, m_2, x)) = \text{Up}(m_1, m_2, e) \quad (m_1 \leq m_2) \]

(LetAssoc) \[ \text{Let}(m_1, m_3, x, \text{Let}(m_2, m_1, y, e_1, e_2), b) = \] \[ \text{Let}(m_2, m_1, y, e_1, \text{Let}(m_1, m_3, x, e_2, b)) \quad (m_2 \leq m_1, y \not\in \text{FV}(b)) \]

(LetrecAssoc) \[ \text{Let}(m_1, m_2, x, \text{Letrec}(f, y, e_1, e_2), b) = \] \[ \text{Letrec}(f, y, e_1, \text{Let}(m_1, m_2, x, e_2, b)) \quad (y \not\in \text{FV}(b)) \]

(LetUp) \[ \text{Let}(m_1, m_3, x, e, \text{Up}(m_2, m_3, b)) = \text{Up}(m_2, m_3, \text{Let}(m_1, m_2, x, e, b)) \quad (m_1 \leq m_2 \leq m_3) \]

Figure 8: Generalized monad laws

4 Transformation Rules

In this section we attempt to motivate our IR, and in particular our choice of monads, by presenting a number of useful transformation laws, which can be proved correct with respect to the denotational semantics. (These proofs are straightforward but tedious, so are omitted here.) Of course, this is by no means a complete set of rules needed by an optimizer; there are many others: both general-purpose and specific to particular operators. Also, as noted earlier, not all valid transformations are improvements.

Figure 8 gives general rules for manipulating monadic expressions. (MonadID1), (MonadID2), and (LetAssoc) are generalizations of the usual laws for a single monad, which can be recovered from these rules by setting \( m_1 = \text{ID} \) in (MonadID1), and setting \( m_1 = m_2 \) in (MonadID2) and (LetAssoc). (LetrecAssoc) is the corresponding associativity rule for Letrecs. (LetUp) permits us to move expressions with weak effects in and out of coercions. The remaining rules let us do housekeeping on coercions.

Figure 9 lists some valid laws for altering execution order. We have full beta reduction for variables bound in the ID monad (BetaID). In general, the order of two bindings can be exchanged if there is no data dependence between them, and if either of them is in the ID monad (ExchangeID) or both are in or below the LIFT monad (ExchangeLIFT). The intuition for the latter rule is that it harmless to reorder two expressions even if one or both may not terminate, because we cannot detect which one causes the non-termination. On the other hand, there is no similar rule for the EXN monad, because we can distinguish different raised exceptions according to the constructor value.
\( (\text{BetaID}) \quad \text{Let}(\text{ID}, m, x, e, b) = b[e/x] \)

\( (\text{ExchangeID}) \quad \text{Let}(m_1, m_3, x_1, e_1, \text{Let}(m_2, m_3, x_2, e_2, b)) = \)
\( \quad \text{Let}(m_2, m_3, x_2, e_2, \text{Let}(m_1, m_3, x_1, e_1, b)) \)
\( \quad (m_1 = \text{ID} \text{ or } m_2 = \text{ID}; x_1 \notin FV(e_2); x_2 \notin FV(e_1)) \)

\( (\text{ExchangeLIFT}) \quad \text{Let}(m_1, m_3, x_1, e_1, \text{Let}(m_2, m_3, x_2, e_2, b)) = \)
\( \quad \text{Let}(m_2, m_3, x_2, e_2, \text{Let}(m_1, m_3, x_1, e_1, b)) \)
\( \quad (m_1 < \text{LIFT}; x_1 \notin FV(e_2); x_2 \notin FV(e_1)) \)

\( (\text{HoistID}) \quad \text{Letrec}(f, x, \text{Let}(\text{ID}, m_2, y, e_1, e_2), b) = \)
\( \quad \text{M}(m, t) = \)
\( \quad \text{Let}(\text{ID}, m, y, e_1, \text{Letrec}(f, x, e_2, b)) \)
\( \quad (f, x \notin FV(e_1)) \)

\( (\text{HoistEXN}) \quad \text{Letrec}(f, x, \text{Let}(m_1, m_2, y, e_1, e_2), \text{App}(f, z)) = \)
\( \quad \text{Let}(m_1, m_2, y, e_1, \text{Letrec}(f, x, e_2, \text{App}(f, z))) \)
\( \quad (m_1 \leq \text{EXN}; x, f \notin FV(e_1)) \)

\( (\text{IfID}) \quad \text{If}(v, \text{Let}(\text{ID}, m, x, e_1, e_2), e_3) = \text{Let}(\text{ID}, m, x, e_1, \text{If}(v, e_2, e_3)) \)
\( \quad (x \notin FV(e_3)) \)

Figure 9: Exchange laws for monadic expressions

they carry. This is the principal point of difference between LIFT and EXN from an optimization standpoint.

Rule (HoistID) states that it always valid to lift a pure expression out of a Letrec (if no data dependence is violated). (HoistEXN) reflects a much stronger property: it is valid to lift a non-terminating or exception-raising expression of a Letrec if the recursive function is guaranteed to be executed at least once. This is the principal advantage of distinguishing EXN from the more general ST monad, for which the transform is not valid. Although the left-hand side of (HoistEXN) may seem a crude way to characterize functions guaranteed to be called at least once, and unlikely to appear in practice, it arises naturally if we systematically introduce loop headers for recursions [1], according to the following law:

\( (\text{Header}) \quad \text{Letrec}(f, x, e, b) : \text{M}(m, t) = \)
\( \quad \text{Let}(\text{ID}, m, f, \text{Abs}(z, \text{Letrec}(f', z, e[f'/f], \text{App}(f', z))), b) \)
\( \quad (f' \notin FV(e)) \)

Finally, we include the rule (IfID) as an example of the flexibility with which ID expressions can be manipulated; there are similar rules for floating ID expressions out of other constructs.

As a (rather artificial) example of the power of these transformations, consider the code in Figure 10. The computation of \( w \) is invariant, so we would like to hoist it above recursive function \( r \). Because the binding for \( w \) is marked as pure and terminating, it can be lifted out of the \( \text{if} \) using (IfID), and can then be exchanged with the pure bindings for \( s \) and \( t \) using (ExchangeID). This positions it to be lifted out of \( r \) using (HoistID). Note that the monad annotations tell us that \( w \) is
let f:(Int -> M(ID,Int * Int)) -> M(ST,Int) =
\lambda g:(Int->M(ID,Int * Int)).
letrec r:Int->M(ST,Int) =
\lambda x:Int.letID t:Int * Int = (x,1)
in letID s:Bool = EqInt(t)
in if s then
  Up(ID,ST,0)
else
  letID w:Int * Int = g(3)
in letID y:Int = Plus(w)
  in letID z:Int * int = (x,y)
in letEXN x':Int = Divide(z)
in letST dummy:() = WriteInt(x')
in r(x')
in r(10)
in let h:Int->M(ID,Int * Int) = \lambda p:Int.(p,p)
in f(h)

Figure 10: Example of intermediate code, presented in an obvious concrete analogue of the abstract syntax.

pure and terminating even though it invokes the unknown function g, which is actually bound to h.

The example also exposes the limitations of monomorphic effects: if f were also applied to an impure function, then g and hence w would be marked as impure, and the binding for w could not be hoisted. In practice, it might be desirable to clone separate copies of f, specialized according to the effectfulness of their g argument. Worse yet, consider a function that is naturally parametric in its effect, such as map. Such a function will always be pessimistically annotated with an effect reflecting the most-effectful function passed to it within the program. The obvious solution is to give functions like map a generic type abstracted over a monad variable, analogous to an effect variable in the system of Talpin and Jouvelot [14]. We believe our system can be extended to handle such generic types, but we have not examined the semantic issues involved in detail.

5 Monad Inference

It would be possible to translate source programs into type-correct IR programs by simply assuming that every expression falls into the maximally-effectful monad (ST in our case). Every source Let would become a LetST, every variable and constant would be coerced into ST, and every primitive would return a value in ST. Peyton Jones et al. [11] suggest performing such a translation, and then using the monad laws (analogous to those in Figure 8) and the worker-wrapper transform [13] to simplify the result, hopefully resulting in some less-effectful expression bindings. The main objection to this approach is that it doesn't allow calls to unknown functions (for which worker-wrapper doesn't apply) to return non-ST results. For example, in the code of Figure 10, no local
\[
E \vdash v : \text{Bool} \quad E \vdash e_1 \Rightarrow e'_1 : M(m, t) \quad E \vdash e_2 \Rightarrow e'_2 : M(m, t)
\]

\[
E \vdash \text{If}(v, e_1, e_2) \Rightarrow \text{If}(v, e'_1, e'_2) : M(m, t)
\]

\[
E \vdash e_1 \Rightarrow e'_1 : M(m_1, t_1) \quad E \vdash \{x : t_1\} \vdash e_2 \Rightarrow e'_2 : M(m_2, t_2) \quad (m_1 \leq m_2)
\]

\[
E \vdash \text{Let}(x, e_1, e_2) \Rightarrow \text{Let}(m_1, m_2, x : t_1, e'_1, e'_2) : M(m_2, t_2)
\]

\[
E \vdash e \Rightarrow e' : M(m_1, t) \quad m_1 \leq m_2
\]

\[
E \vdash e \Rightarrow \text{Up}(m_1, m_2, e') : M(m_2, t)
\]

Figure 11: Selected translation rules

syntactic analysis could discover that argument function g is pure and terminating.

To obtain better control over effects, we have developed an inference algorithm for computing the minimal monadic effect of each subexpression in a program. Pure, provably terminating expressions are placed in ID, pure but potentially non-terminating expressions in LIFT, and so forth. The algorithm deals with the latent monadic effects in functions, by recording them in the result types. As an example, it produces the annotations shown in Figure 10.

The input to the algorithm is an untyped program in the source language; the output is a program in the typed IR. The algorithm performs ordinary type inference, monad inference, and program translation simultaneously. The type inference aspect uses unification in a completely conventional way, except that unifying the codomain \(\text{mtyps}\) of two arrow types requires unifying their monad components as well as their \(\text{vtyps}\) components. We therefore omit a detailed description of \(\text{vtyp}\) unification.

The translation aspect is also quite straightforward. We can turn each typing rule in Figure 3 into a translation rule simply by recording the inferred type and monad information in the appropriate annotation slots of the output and combining the translations of subterms in the obvious manner. As examples, Figure 11 shows the translation rules corresponding to the typing rules for If, Let, and Up. In cases where a monad or type appears in the translation output, such as \(m_1\) and \(t_1\) in the Let rule, a fresh monad or type variable is created and inserted in the output for subsequent instantiation. Type variables are instantiated by unification; the method of instantiating monad variables is described below.

Excluding the rule for Up, the resulting translation rules form a deterministic, syntax-directed algorithm for translation, giving an output program with exactly the same term structure as the input. However, the resulting program may not obey the monadic constraints in the typing rules. Consider, for example, the source term \(\text{If}(x, \text{Val} y, \text{Raise} z)\). Since \(\text{Val} y\) is a value, its translation is in the ID monad, whereas the translation of \(\text{Raise} z\) must be in the EXN or ST monad. To glue together these subterm translations we must insert a coercion around the translation of the \(\text{Val}\) term. The “translation” rule for Up is really a coercion insertion rule, which serves exactly this purpose; it adds the necessary flexibility to the system to permit all monad constraints to be met.
Since this rule can be applied to any subexpression, it adds a problematic element of nondeterminism to the system. Our solution is to insert a (single) Up coercion around every subexpression, and rely on a postprocessing step to remove unneeded coercions using the (IdentUp) rule. (The complete Standard ML code for the translation routine is given in Appendix A.)

The final consideration is how to record and resolve constraints on the monad variables. Such constraints are introduced explicitly by the side conditions in the Let, Letrec, and Up rules, implicitly by the equating of monads from subexpressions in the If and Handle rules, and (even more) implicitly as a result of ordinary unification of arrow types, which mention monads in their codomains. The side-condition constraints are all inequalities of the form \( m_1 \geq m_2 \), where \( m_1 \) is a monad variable and \( m_2 \) is a variable or an explicit monad. The implicit constraints are all equalities \( m_1 = m_2 \); for uniformity, we replace these by a pair of inequalities: \( m_1 \geq m_2 \) and \( m_2 \geq m_1 \). We collect constraints as a side-effect of the translation process, simply by adding them to a global list.

It is very common for there to be circularities among the monad constraints. To solve the constraint system, we think of it as a directed graph with a node for each monad and monad variable, and an edge from \( m_1 \) to \( m_2 \) for each constraint \( m_1 \geq m_2 \). We then partition the graph into its strongly connected components, and sort the components into reverse topological order. We process one component at a time, in this order. Since \( \geq \) is a partial order, all the nodes in a given component must be assigned the same monad; once this has been determined, it is assigned to all the variables in the component before proceeding to the next component. To determine the minimum possible correct assignment for a component, we consult all the edges from nodes in that component to nodes outside the component; because of the order of processing, these nodes must already have received a monad assignment. The maximum of these assignments is the minimum correct assignment for this component. If there are no such edges, the minimum correct assignment is \( ID \). This algorithm is linear in the number of constraints, and hence in the size of the source program.

To summarize, we perform monad inference by first translating the source program into a form padded with coercion operators and annotated with monad variables, meanwhile collecting constraints on these variables, and then solving the resulting constraint system to fill in the variables in the translated program. The resulting program will contain many null coercions of the form \( \text{Up}(m, m, e) \); these can be removed by a single postprocessing pass.

Our algorithm is very similar to a that of Talpin and Jouvelot [14], restricted to a monomorphic source language. Both algorithms generate essentially the same sets of constraints. Talpin and Jouvelot apparently solve the constraints using unification; the full details of the unification algorithm are not given. It would be natural to extend our algorithm to handle Hindley-Milner polymorphism for both types and monads in the Talpin-Jouvelot style. The idea is to generalize all free type and effect variables in \texttt{let} definitions and allow different uses of the bound identifier to instantiate these in different ways. In particular, parametric functions like \texttt{map} could be used with many different monads, without one use "polluting" the others. (Note that functions not wholly parametric in their effects would place a minimum effect bound on permissible instantiations for
monad variables.) This form of monad polymorphism seems desirable even in the absence of type polymorphism (e.g., resulting from explicit monomorphization [17]).

In whole-program compilation, the complete set of effect instantiations would be known. This set could be used to put an upper effect bound on monad variables within definition bodies and hence determine what transformations are legal there. Alternatively, it could be used to guide the generation of effect-specific clones as suggested in the previous section. Generalization of effect variables would also support safe separate compilation, though drawbacks would remain: in the absence of complete information about uses of a definition, any variable monad in the body of the definition must be treated as ST, the most “effectful” monad, for the purposes of performing transformations within the body.

6 Status and Conclusions

We believe our approach has the merits of simplicity and reasonable effectiveness. We have implemented the monad inference algorithm for an extended version of the IR described here, which supports full Standard ML; we are currently measuring its effectiveness using the backend of our RML compiler system [17].

Acknowledgements

We have benefitted from conversations with John Launchbury and Dick Kieburtz, and from exposure to the ideas in their unpublished papers [6, 7]. The comments of the anonymous referees also motivated us to clarify the relationship of our algorithm with the existing work of Talpin and Jouvelot.

References


Appendix: Code for monadic inference translation

```haskell
fun unify_typ (M(ma,ta),M(mb,tb)) =
    (bound_monad(ma,mb); bound_monad(mb,ma); unify_vtyp(ta,tb))

and unify_vtyp (a:vtyp,b:vtyp) = ...unify_typ...

and bound_monad (ma:monad,mb:monad) = ...

fun type_value (env:id -> vtyp) (v:value) : typ = ...

fun wrap(e : exp.t as M(m,vt)) : exp * typ =
    let val m' = new_monad()
    in bound_monad(m',m);
        (Up(m,m',e),M(m',vt))
    end

fun translate_exp (env:id -> vtyp) (e: exp) : exp * typ =
    case e of
        Source.Val v => let val t' = type_value env v
                        in wrap(Val v, M(ID,t'))
                    end
    | Source.Abs(x,e) =>
        let val t = new_vtyp()
            val (e',t') = translate_exp (extend env (x,t)) e
            in wrap(Abs((x,t),e'),M(ID,Arrow(t,t')))
        end
    | Source.App(v1,v2) =>
        let val t = new_vtyp() and u = new_typ()
            val t1 = type_value env v1
            val t2 = type_value env v2
            in unify_vtyp(Arrow(t,u), t1');
                unify_vtyp(t,t2');
                wrap(App(v1,v2),u)
            end
```

18
| Source.If(v,el,e2) =>
  let val t' = type_value env v
  val (el',t1') = translate_exp env el
  val (e2',t2') = translate_exp env e2
  in unify_vtyp(t',Bool);
  unify_typ(t1',t2');
  wrap(If(v,el',e2'),t1')
end

| Source.Let(x,el,e2) =>
  let val (el',t1' as M(ml',vtl')) = translate_exp env el
  val (e2',t2' as M(m2',vt2')) = translate_exp (extend env (x,vtl')) e2
  in bound_monad(m2',ml');
  wrap(Let(ml,m2',(x,vtl'),el',e2'),t2')
end

| Source.Letrec(f,x,el,e2) =>
  let val t = new_vtyp() and u as M(um,uvt) = new_typ()
  val (el',t1') = translate_exp (extend (extend env (f.Arrow(t,u))) (x,t)) el
  val (e2',t2') = translate_exp (extend env (f,Arrow(t,u))) e2
  in unify_typ (tl',u);
  bound_monad(um,LIFT);
  wrap(Letrec((f,Arrow(t,u)),(x,t),el',e2'), t2')
end

| Source.Tuple vs =>
  let val ts = map (type_value env) vs
  in wrap(Tuple vs,M(ID,Tup ts))
end

| Source.Proj(i,n,v) =>
  let val t' = type_value env v
  fun upto (x,y) = if x > y then [] else x::(upto (x+l,y))
  val vts = map new_vtyp (upto (0,n-l))
  val t = List.nth(vts,i) handle Subscript => raise Bad "Proj index"
  in unify_vtyp(t'.Tup(vts));
  wrap(Proj(i,n,v),M(ID,t))
end

| Source.Raise (v) =>
  let val vt = new_vtyp()
  val t = M(EXN,vt)
  val t' = type_value env v
  in unify_vtyp(t'.Exn);
  wrap(Raise(t,v),t)
end

| Source.Handle(e,v) =>
  let val u as M(um,uvt) = new_typ()
  val (e',t') = translate_exp env e
  val vt' = type_value env v
  in unify_typ(u,t');
  bound_monad(um,EXN);
  unify_vtyp(vt',Arrow(Exn,t'));
  wrap(Handle(m,e',v),t')
end