Bridging the gulf: a common intermediate language for ML and Haskell

Simon Peyton Jones
University of Glasgow and Oregon Graduate Institute

John Launchbury
Oregon Graduate Institute

Mark Shields
University of Glasgow and 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 [1992a]), but, with one exception[1], 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, $\mathcal{L}_1$, is mathematically simple, but cannot be compiled well (Section 3). An alternative design, $\mathcal{L}_2$, adds operational significance to unpointed types, by guaranteeing that a variable of unpointed type is evaluated (Section 4); this means $\mathcal{L}_2$ can be compiled well, but weakens its theory.
- We identify an interaction between unpointed types, polymorphism, and recursion in $\mathcal{L}_1$ (Section 3.5). Interestingly, the problem turns out to be more easily solved in $\mathcal{L}_2$ than $\mathcal{L}_1$ (Section 4.2).

To appear in the ACM Symposium on Principles of Programming Languages (POPL'98)

[1] Personal communication, Nick Benton, Persimmon IT Ltd, 1997.

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 them 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 [1993]).* 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 $\mathcal{L}_1$, 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[2]

$$\mathcal{E}[\![e_1\ e_2]\!]\rho = \begin{array}{ll} (\mathcal{E}[\![e_1]\!]\rho)\ b, & \text{if } a = b_\perp \\ \perp, & \text{if } a = \perp \\ \text{where } a = \mathcal{E}[\![e_2]\!]\rho & \end{array}$$

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, $\mathcal{L}_1$.

Figure 1 gives the syntax and type rules for $\mathcal{L}_1$. 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 $M\ \tau$ is the type of M-computations returning a value of type τ, where M is drawn from a fixed family of monads. The syntactic forms $\texttt{let}_M$ and $\texttt{ret}_M$ are

[2] We use the following standard notation. If T is a complete partial order (CPO), then the CPO $T_\perp$, pronounced "T lifted", is defined thus: $T_\perp = \{a_\perp \mid a \in T\} \cup \{\perp\}$, with the obvious ordering.

$$
\begin{array}{rrcl}
\mathit{Types}\ \tau,\rho & & ::= & \texttt{Int} \mid \tau_1\texttt{->}\tau_2 \mid \texttt{()} \mid \texttt{(}\tau_1\texttt{,}\tau_2\texttt{)} \\
& & \mid & \texttt{Ref}\ \tau \mid M\ \tau \\
\mathit{Terms} & e & ::= & x \mid k \mid e_1\ e_2 \mid \texttt{\\}x\texttt{:}\tau\texttt{.}e \mid \texttt{(}e_1\texttt{,}e_2\texttt{)} \\
& & \mid & \texttt{let}\ x\texttt{:}\tau\texttt{=}e_1\ \texttt{in}\ e_2 \\
& & \mid & \texttt{letrec}\ x\texttt{:}\tau\texttt{=}e_1\ \texttt{in}\ e_2 \\
& & \mid & \texttt{let}_M\ x\texttt{:}\tau\ \texttt{<-}\ e_1\ \texttt{in}\ e_2 \mid \texttt{ret}_M\ e \\
\mathit{Constants} & k & ::= & \texttt{fst} \mid \texttt{snd} \mid \texttt{new} \mid \texttt{rd} \mid \texttt{wr} \mid \texttt{liftToST} \\
& & \mid & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \mid \texttt{+} \mid \texttt{-} \mid \ldots \\
\mathit{Monads} & M & ::= & \texttt{Lift} \mid \texttt{ST}
\end{array}
$$

$$
\begin{array}{ll}
(VAR) & \dfrac{x:\tau \in \Gamma}{\Gamma \vdash x:\tau} \\[2ex]
(PAIR) & \dfrac{\Gamma \vdash e_1:\tau_1 \quad \Gamma \vdash e_2:\tau_2}{\Gamma \vdash \texttt{(}e_1\texttt{,}e_2\texttt{)} : \texttt{(}\tau_1\texttt{,}\tau_2\texttt{)}} \\[2ex]
(APP) & \dfrac{\Gamma \vdash e_1:\tau\texttt{->}\rho \quad \Gamma \vdash e_2:\tau}{\Gamma \vdash e_1\ e_2 : \rho} \\[2ex]
(LAM) & \dfrac{\Gamma, x\texttt{:}\tau \vdash e:\rho}{\Gamma \vdash \lambda x\texttt{:}\tau.e : \tau\texttt{->}\rho} \\[2ex]
(LET) & \dfrac{\Gamma \vdash e_1:\tau \quad \Gamma, x\texttt{:}\tau \vdash e_2:\rho}{\Gamma \vdash \texttt{let}\ x\texttt{:}\tau\texttt{=}e_1\ \texttt{in}\ e_2 : \rho} \\[2ex]
(REC) & \dfrac{\Gamma, x:\tau \vdash e_1:\tau \quad \Gamma, x:\tau \vdash e_2:\rho \quad \text{...plus a side condition...}}{\Gamma \vdash \texttt{letrec}\ x\texttt{:}\tau\texttt{=}e_1\ \texttt{in}\ e_2 : \rho} \\[2ex]
(LETM) & \dfrac{\Gamma \vdash e_1 : M\ \tau_1 \quad \Gamma, x:\tau_1 \vdash e_2 : M\ \tau_2}{\Gamma \vdash \texttt{let}_M\ x\texttt{:}\tau_1\ \texttt{<-}\ e_1\ \texttt{in}\ e_2 : M\ \tau_2} \\[2ex]
(RET) & \dfrac{\Gamma \vdash e:\tau}{\Gamma \vdash \texttt{ret}_M\ e : M\ \tau} \\[2ex]
(FST) & \Gamma \vdash \texttt{fst} : \texttt{(}\tau_1\texttt{,}\tau_2\texttt{)}\ \texttt{->}\ \tau_1 \\
(SND) & \Gamma \vdash \texttt{snd} : \texttt{(}\tau_1\texttt{,}\tau_2\texttt{)}\ \texttt{->}\ \tau_2 \\
(PLUS) & \Gamma \vdash \texttt{+} : \texttt{Int -> Int -> Int} \\
(NEW) & \Gamma \vdash \texttt{new} : \tau\ \texttt{-> ST (Ref}\ \tau\texttt{)} \\
(RD) & \Gamma \vdash \texttt{rd} : \texttt{Ref}\ \tau\ \texttt{-> ST}\ \tau \\
(WR) & \Gamma \vdash \texttt{wr} : \texttt{Ref}\ \tau\ \texttt{->}\ \tau\ \texttt{-> ST ()} \\
(LIFT) & \Gamma \vdash \texttt{liftToST} : \texttt{Lift}\ \tau\ \texttt{-> ST}\ \tau
\end{array}
$$

Figure 1: Syntax and type rules for $\mathcal{L}_1$

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 $\mathcal{L}_1$ should have a trivial semantics. We discuss the reasons for this decision in Section 3.4.

Figure 2 gives the semantics of $\mathcal{L}_1$. The semantic function $\mathcal{T}$ gives the meaning of types. If it looks somewhat boring, that is the point! The function arrow in $\mathcal{L}_1$ is interpreted by function arrow in the underlying category of complete partial orders ($\mathcal{CPO}$), product is interpreted by (categorical, i.e.

$$
\begin{array}{rcl}
\mathcal{T} : \mathit{Type} & \to & \mathcal{CPO} \\
\mathcal{T}[\![\texttt{Int}]\!] & = & \mathcal{Z} \\
\mathcal{T}[\![\tau_1\texttt{->}\tau_2]\!] & = & \mathcal{T}[\![\tau_1]\!] \to \mathcal{T}[\![\tau_2]\!] \\
\mathcal{T}[\![\texttt{(}\tau_1\texttt{,}\tau_2\texttt{)}]\!] & = & \mathcal{T}[\![\tau_1]\!] \times \mathcal{T}[\![\tau_2]\!] \\
\mathcal{T}[\![\texttt{()}]\!] & = & \mathbf{1} \\
\mathcal{T}[\![\texttt{Lift}\ \tau]\!] & = & \mathcal{T}[\![\tau]\!]_\perp \\
\mathcal{T}[\![\texttt{ST}\ \tau]\!] & = & \mathbf{State} \to (\mathcal{T}[\![\tau]\!] \times \mathbf{State})_\perp \\
\mathcal{T}[\![\texttt{Ref}\ \tau]\!] & = & \mathcal{N} \\
\mathbf{State} & = & \mathcal{N} \hookrightarrow \bigcup_\tau \mathcal{T}[\![\tau]\!]
\end{array}
$$

$$
\begin{array}{rcl}
\multicolumn{3}{l}{\mathcal{E} : \mathit{Term}_\tau \to \mathit{Env} \to \mathcal{T}[\![\tau]\!]} \\
\mathcal{E}[\![x]\!]\rho & = & \rho(x) \\
\mathcal{E}[\![k]\!]\rho & = & k \\
\mathcal{E}[\![e_1\ e_2]\!]\rho & = & (\mathcal{E}[\![e_1]\!]\rho)\ (\mathcal{E}[\![e_2]\!]\rho) \\
\mathcal{E}[\![\texttt{\\}x\texttt{.}e]\!]\rho & = & \lambda y.\mathcal{E}[\![e]\!]\rho[x := y] \\
\mathcal{E}[\![\texttt{(}e_1\texttt{,}e_2\texttt{)}]\!]\rho & = & (\mathcal{E}[\![e_1]\!]\rho, \mathcal{E}[\![e_2]\!]\rho) \\
\mathcal{E}[\![\texttt{let}\ x\texttt{:}\tau\texttt{=}e_1\ \texttt{in}\ e_2]\!]\rho & = & \mathcal{E}[\![e_2]\!]\rho[x := \mathcal{E}[\![e_1]\!]\rho] \\
\mathcal{E}[\![\texttt{letrec}\ x\texttt{:}\tau\texttt{=}e_1\ \texttt{in}\ e_2]\!]\rho & = & \mathcal{E}[\![e_2]\!](\mathit{rec}[\![x,e_1]\!]\rho) \\
\mathcal{E}[\![\texttt{let}_M\ x\texttt{:}\tau\ \texttt{<-}\ e_1\ \texttt{in}\ e_2]\!]\rho & = & \mathit{bind}_M\ (\mathcal{E}[\![e_1]\!]\rho) \\
& & \qquad (\lambda y.\mathcal{E}[\![e_2]\!]\rho[x := y]) \\
\mathcal{E}[\![\texttt{ret}_M\ e]\!]\rho & = & \mathit{unit}_M\ (\mathcal{E}[\![e]\!]\rho) \\[1ex]
\mathit{rec}[\![x,e_1]\!]\rho & = & \mathit{fix}(\lambda\rho'.\rho[x := \mathcal{E}[\![e_1]\!]\rho'])
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{fst}\ (a,b) & = & a & \\
\mathit{snd}\ (a,b) & = & b & \\[1ex]
\mathit{bind}_{\mathit{Lift}}\ m\ k & = & \perp, & \text{if } m = \perp \\
& & k\ a, & \text{if } m = a_\perp \\
\mathit{unit}_{\mathit{Lift}}\ x & = & x_\perp & \\[1ex]
\mathit{bind}_{\mathit{ST}}\ m\ k\ s & = & \perp, & \text{if } m\ s = \perp \\
& & k\ r\ s', & \text{if } m\ s = (r,s')_\perp \\
\mathit{unit}_{\mathit{ST}}\ m\ s & = & (m,s)_\perp & \\
\mathit{new}\ v\ s & = & (r, s[r \mapsto v])_\perp & \text{where } r \notin \mathit{dom}(s) \\
\mathit{rd}\ r\ s & = & (s\ r, s)_\perp, & \text{if } r \in \mathit{dom}(s) \\
& & \perp, & \text{otherwise} \\
\mathit{wr}\ r\ v\ s & = & ((), s[r \mapsto v])_\perp, & \text{if } r \in \mathit{dom}(s) \\
& & \perp, & \text{otherwise} \\
\mathit{liftToST}\ m\ s & = & (r,s)_\perp, & \text{if } m = r_\perp \\
& & \perp, & \text{otherwise}
\end{array}
$$

Figure 2: Semantics of $\mathcal{L}_1$

un-lifted) product, and integers are interpreted by the integers. (If $\mathcal{L}_1$ 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, $\mathcal{N}$) to their contents.

The semantic function $\mathcal{E}$ 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 β and η are valid with respect to the semantics, and that monadic expressions admit a number of standard transformations, given in Figure 3.

$$
\begin{array}{llcll}
(M1) & \texttt{let}_M\ x \texttt{<-} \texttt{ret}_M\ e\ \texttt{in}\ b & = & \texttt{let}\ x{:}\tau \texttt{=} e\ \texttt{in}\ b & \\
(M2) & \texttt{let}_M\ x \texttt{<-} (\texttt{let}_M\ y \texttt{<-} e_1\ \texttt{in}\ e_2)\ \texttt{in}\ b & = & \texttt{let}_M\ y \texttt{<-} e_1\ \texttt{in}\ (\texttt{let}_M\ x \texttt{<-} e_2\ \texttt{in}\ b) & y \notin fv(b) \\
(M3) & \texttt{let}_M\ x \texttt{<-} (\texttt{let}\ y \texttt{=} e_1\ \texttt{in}\ e_2)\ \texttt{in}\ b & = & \texttt{let}\ y \texttt{=} e_1\ \texttt{in}\ (\texttt{let}_M\ x \texttt{<-} e_2\ \texttt{in}\ b) & y \notin fv(b) \\
(M4) & \texttt{let}_M\ x \texttt{<-} (\texttt{letrec}\ y \texttt{=} e_1\ \texttt{in}\ e_2)\ \texttt{in}\ b & = & \texttt{letrec}\ y \texttt{=} e_1\ \texttt{in}\ (\texttt{let}_M\ x \texttt{<-} e_2\ \texttt{in}\ b) & y \notin fv(b) \\
(M5) & \texttt{let}_M\ x \texttt{<-} e\ \texttt{in}\ \texttt{ret}_M\ x & = & e & \\
(M6) & \texttt{let}\ x \texttt{=} e\ \texttt{in}\ \texttt{ret}_M\ b & = & \texttt{ret}_M\ (\texttt{let}\ x \texttt{=} e\ \texttt{in}\ b) &
\end{array}
$$

Figure 3: Monad transformations

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, `Int`, is interpreted by the *unpointed* CPO of integers, $\mathcal{Z}$. That is, if an expression has type `Int`, then it denotes an integer, and *cannot denote a non-terminating computation*. How, then, do we express the type of possibly-diverging integer-valued computations? 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 `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 `\x:Int.`e expresses that `x` cannot possibly be $\perp$, and so is a suitable translation of a lambda abstraction from a call-by-value language. On the other hand `\x:Lift Int.`e expresses that `x` might perhaps be $\perp$, 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 & Fukushima [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 Int` denotes a pure (side-effect free), albeit possibly-divergent, computation; on the other hand, and expression of type `ST Int` denotes a computation that might diverge[3], 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] `ST` combines lifting with state. It would be possible to separate the two, as we discuss in Section 7.

$$
\begin{array}{llcl}
\text{Types} & S,T & ::= & \mathbf{Int} \mid () \mid S * T \mid S \to T \mid \mathbf{Ref}\ S \\
\textit{Haskell only} & & \mid & \mathbf{ST}\ S \\
\text{Terms} & M,N & ::= & x \mid i \mid M\ N \mid \lambda x{:}T.M \mid M + N \\
 & & \mid & \mathbf{letrec}\ x{:}T = M\ \mathbf{in}\ N \\
 & & \mid & \mathbf{let}\ x{:}T = M\ \mathbf{in}\ N \\
 & & \mid & \mathbf{pair}\ M\ N \mid \mathbf{fst}\ M \mid \mathbf{snd}\ M \\
 & & \mid & \mathbf{new}\ M \mid \mathbf{rd}\ M \mid \mathbf{wr}\ M\ N \\
\textit{Haskell only} & & \mid & \mathbf{let_{ST}}\ x{:}T \leftarrow M\ \mathbf{in}\ N \mid \mathbf{ret_{ST}}\ M \\
\text{Integers} & i & ::= & 0 \mid 1 \mid 2 \mid \ldots
\end{array}
$$

"ML" constants

$$
\begin{array}{rcl}
\mathbf{new} & : & \forall\alpha.\alpha \to \mathbf{Ref}\ \alpha \\
\mathbf{rd} & : & \forall\alpha.\mathbf{Ref}\ \alpha \to \alpha \\
\mathbf{wr} & : & \forall\alpha.\mathbf{Ref}\ \alpha \to \alpha \to ()
\end{array}
$$

"Haskell" constants

$$
\begin{array}{rcl}
\mathbf{new} & : & \forall\alpha.\alpha \to \mathbf{ST}\ (\mathbf{Ref}\ \alpha) \\
\mathbf{rd} & : & \forall\alpha.\mathbf{Ref}\ \alpha \to \mathbf{ST}\ \alpha \\
\mathbf{wr} & : & \forall\alpha.\mathbf{Ref}\ \alpha \to \alpha \to \mathbf{ST}\ ()
\end{array}
$$

Figure 4: Syntax of $\mathcal{S}$

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

Before discussing its design any further, we first emphasise $\mathcal{L}_1$'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, $\mathcal{S}$. We regard $\mathcal{S}$ as 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**, **fst**, **snd** have the same (obvious) $\mathcal{S}$ types in both interpretations. The constants **new**, **rd**, **wr** create, read, and 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 $\mathcal{S}$ also includes

$$
\begin{aligned}
\mathcal{M}\llbracket \mathbf{Int} \rrbracket &= \texttt{Int} \\
\mathcal{M}\llbracket S * T \rrbracket &= \texttt{(}\mathcal{M}\llbracket S \rrbracket\texttt{,}\mathcal{M}\llbracket T \rrbracket\texttt{)} \\
\mathcal{M}\llbracket () \rrbracket &= \texttt{()} \\
\mathcal{M}\llbracket S \to T \rrbracket &= \mathcal{M}\llbracket S \rrbracket \texttt{ -> ST } \mathcal{M}\llbracket T \rrbracket \\
\mathcal{M}\llbracket \mathbf{Ref}\ S \rrbracket &= \texttt{Ref (}\mathcal{M}\llbracket S \rrbracket\texttt{)} \\
\\
\mathcal{M}\llbracket x \rrbracket &= \texttt{ret}_{\texttt{ST}}\ x \\
\mathcal{M}\llbracket i \rrbracket &= \texttt{ret}_{\texttt{ST}}\ i \\
\mathcal{M}\llbracket M\ N \rrbracket &= \texttt{let}_{\texttt{ST}}\ f \texttt{ <- } \mathcal{M}\llbracket M \rrbracket \texttt{ in} \\
&\ \texttt{let}_{\texttt{ST}}\ a \texttt{ <- } \mathcal{M}\llbracket N \rrbracket \texttt{ in} \\
&\ f\ a \\
\mathcal{M}\llbracket \lambda x{:}T.M \rrbracket &= \texttt{ret}_{\texttt{ST}}\ (\texttt{\textbackslash} x{:}\mathcal{M}\llbracket T \rrbracket\texttt{.}\mathcal{M}\llbracket M \rrbracket) \\
\mathcal{M}\llbracket \mathbf{let}\ x{:}T = M\ \mathbf{in}\ N \rrbracket & \\
&= \texttt{let}_{\texttt{ST}}\ x{:}\mathcal{M}\llbracket T \rrbracket \texttt{ <- } \mathcal{M}\llbracket M \rrbracket \texttt{ in } \mathcal{M}\llbracket N \rrbracket \\
\mathcal{M}\llbracket \mathbf{letrec}\ f{:}S \to T = \lambda x : S.M\ \mathbf{in}\ N \rrbracket & \\
&= \texttt{letrec } f{:}\mathcal{M}\llbracket S \to T \rrbracket \texttt{ = \textbackslash} x{:}\mathcal{M}\llbracket S \rrbracket.\mathcal{M}\llbracket M \rrbracket \texttt{ in } \mathcal{M}\llbracket N \rrbracket \\
\mathcal{M}\llbracket \mathbf{pair}\ M\ N \rrbracket &= \texttt{let}_{\texttt{ST}}\ a \texttt{ <- } \mathcal{M}\llbracket M \rrbracket \texttt{ in} \\
&\ \texttt{let}_{\texttt{ST}}\ b \texttt{ <- } \mathcal{M}\llbracket N \rrbracket \texttt{ in} \\
&\ \texttt{ret}_{\texttt{ST}}\ \texttt{(}a\texttt{,}b\texttt{)} \\
&\ \ldots \text{and similarly } \mathbf{wr}, + \\
\mathcal{M}\llbracket \mathbf{fst}\ M \rrbracket &= \texttt{let}_{\texttt{ST}}\ a \texttt{ <- } \mathcal{M}\llbracket M \rrbracket \texttt{ in} \\
&\ \texttt{ret}_{\texttt{ST}}\ \texttt{fst}\ a \\
&\ \ldots \text{and similarly } \mathbf{snd}, \mathbf{new}, \mathbf{rd}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{H}\llbracket \mathbf{Int} \rrbracket &= \texttt{Lift Int} \\
\mathcal{H}\llbracket S * T \rrbracket &= \texttt{Lift (}\mathcal{H}\llbracket S \rrbracket\texttt{,}\mathcal{H}\llbracket T \rrbracket\texttt{)} \\
\mathcal{H}\llbracket () \rrbracket &= \texttt{Lift ()} \\
\mathcal{H}\llbracket S \to T \rrbracket &= \mathcal{H}\llbracket S \rrbracket \texttt{ -> } \mathcal{H}\llbracket T \rrbracket \\
\mathcal{H}\llbracket \mathbf{ST}\ T \rrbracket &= \texttt{ST (}\mathcal{H}\llbracket T \rrbracket\texttt{)} \\
\mathcal{H}\llbracket \mathbf{Ref}\ S \rrbracket &= \texttt{Lift (Ref (}\mathcal{H}\llbracket S \rrbracket\texttt{))} \\
\\
\mathcal{H}\llbracket x \rrbracket &= x \\
\mathcal{H}\llbracket i \rrbracket &= \texttt{ret}_{\texttt{Lift}}\ i \\
\mathcal{H}\llbracket M\ N \rrbracket &= \mathcal{H}\llbracket M \rrbracket\ \mathcal{H}\llbracket N \rrbracket \\
\mathcal{H}\llbracket \lambda x{:}T.M \rrbracket &= \texttt{\textbackslash} x{:}\mathcal{H}\llbracket T \rrbracket\texttt{.}\mathcal{H}\llbracket M \rrbracket \\
\mathcal{H}\llbracket \mathbf{let}\ x{:}T = M\ \mathbf{in}\ N \rrbracket &= \texttt{let } x{:}\mathcal{H}\llbracket T \rrbracket \texttt{ = } \mathcal{H}\llbracket M \rrbracket \texttt{ in } \mathcal{H}\llbracket N \rrbracket \\
\mathcal{H}\llbracket \mathbf{letrec}\ x{:}T = M\ \mathbf{in}\ N \rrbracket & \\
&= \texttt{letrec } x{:}\mathcal{H}\llbracket T \rrbracket \texttt{ = } \mathcal{H}\llbracket M \rrbracket \texttt{ in } \mathcal{H}\llbracket N \rrbracket \\
\mathcal{H}\llbracket \mathbf{pair}\ M\ N \rrbracket &= \texttt{ret}_{\texttt{Lift}}\ \texttt{(}\mathcal{H}\llbracket M \rrbracket\texttt{,}\mathcal{H}\llbracket N \rrbracket\texttt{)} \\
\mathcal{H}\llbracket M + N \rrbracket &= \texttt{let}_{\texttt{Lift}}\ a \texttt{ <- } \mathcal{H}\llbracket M \rrbracket \texttt{ in} \\
&\ \texttt{let}_{\texttt{Lift}}\ b \texttt{ <- } \mathcal{H}\llbracket N \rrbracket \texttt{ in} \\
&\ \texttt{ret}_{\texttt{Lift}}\ \texttt{(+ } a\ b\texttt{)} \\
\mathcal{H}\llbracket \mathbf{fst}\ M \rrbracket &= \texttt{let}_{\texttt{Lift}}\ a \texttt{ <- } \mathcal{H}\llbracket M \rrbracket \texttt{ in fst } a \\
&\ \ldots \text{similarly } \mathbf{snd} \\
\mathcal{H}\llbracket \mathbf{wr}\ M\ N \rrbracket &= \texttt{let}_{\texttt{ST}}\ a \texttt{ <- liftToST } \mathcal{H}\llbracket M \rrbracket \texttt{ in} \\
&\ \texttt{wr}\ a\ \mathcal{H}\llbracket N \rrbracket \\
&\ \ldots \text{similarly } \mathbf{new}, \mathbf{rd} \\
\mathcal{H}\llbracket \mathbf{let}_{\mathbf{ST}}\ x{:}T \leftarrow M\ \mathbf{in}\ N \rrbracket & \\
&= \texttt{let}_{\texttt{ST}}\ x{:}\mathcal{H}\llbracket T \rrbracket \texttt{ <- } \mathcal{H}\llbracket M \rrbracket \texttt{ in } \mathcal{H}\llbracket N \rrbracket \\
\mathcal{H}\llbracket \mathbf{ret}_{\mathbf{ST}}\ M \rrbracket &= \texttt{ret}_{\texttt{ST}}\ \mathcal{H}\llbracket M \rrbracket
\end{aligned}
$$

Figure 5: Translations of "ML" and "Haskell" into $\mathcal{L}_1$

$\mathbf{let}_{\mathbf{ST}}$ and $\mathbf{ret}_{\mathbf{ST}}$, 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 $\mathcal{S}$ into $\mathcal{L}_1$:

- The "ML" translation, $\mathcal{M}$[4], 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 $\mathcal{M}\llbracket \Gamma \rrbracket \vdash \mathcal{M}\llbracket e \rrbracket : \texttt{ST}\ \mathcal{M}\llbracket \tau \rrbracket$.

  The rule for application uses $\texttt{let}_{\texttt{ST}}$ 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 $\texttt{ret}_{\texttt{ST}}$. 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 $\mathcal{H}$ necessarily return a `Lift` computation: if the Haskell type system concludes that $\Gamma \vdash e : \tau$ then $\mathcal{H}\llbracket \Gamma \rrbracket \vdash \mathcal{H}\llbracket e \rrbracket : \mathcal{H}\llbracket \tau \rrbracket$.

  $\mathcal{H}$ translates Haskell's **ST**-monad computations directly into $\mathcal{L}_1$'s `ST` monad, just as you would hope[5]. The only tiresome point is that the first argument of **wr** has source-language type **Ref** τ, and hence has $\mathcal{L}_1$ type `Lift (Ref` $\mathcal{H}\llbracket \tau \rrbracket$`)`. 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 `\x.bot` and `bot`, where `bot` has value $\perp$, 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 $\mathcal{L}_1$ 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 $\mathcal{L}_1$. Why, for example, did we not make the `ST` monad explicit by representing a value of type `ST` τ as a state-transforming function in $\mathcal{L}_1$, and representing $\texttt{let}_{\texttt{ST}}$ and

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

[5] We do not treat the **runST** encapsulator of Launchbury & Peyton Jones [1995] here, but it is easy to do so.

$\texttt{ret}_{\texttt{ST}}$ using the other $\mathcal{L}_1$ forms? For example, instead of the $\mathcal{L}_1$ term

$$\texttt{let}_{\texttt{ST}}\ x\ \texttt{<-}\ e\ \texttt{in}\ b$$

we could write the $\mathcal{L}_1$ term

$$\texttt{bindST}\ e\ (\backslash x.b)$$

where `bindST` is defined (directly in $\mathcal{L}_1$) as follows

$$\texttt{bindST = }\backslash m\ k\ s.\texttt{let}\ p\texttt{=}m\ s\ \texttt{in}\ k\ (\texttt{fst}\ p)\ (\texttt{snd}\ p)$$

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 $\texttt{let}_M$ and $\texttt{ret}_M$ 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 β-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

$$\frac{}{\vdash (\texttt{Lift}\ \tau)\ \text{pointed}}$$

$$\frac{}{\vdash (\texttt{ST}\ \tau)\ \text{pointed}}$$

$$\frac{\vdash \tau_1\ \text{pointed}}{\vdash (\tau_2\ \texttt{->}\ \tau_1)\ \text{pointed}}$$

$$\frac{\vdash \tau_1\ \text{pointed} \quad \vdash \tau_2\ \text{pointed}}{\vdash (\tau_1\texttt{,}\tau_2)\ \text{pointed}}$$

Figure 6: Rules for pointed types

`letrec` can be constructed at any type. But that is not so. Consider

```
letrec x:Int = ...x... in ...
```

Such a recursive definition is plainly nonsense, because `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:

$$(RECa)\quad \frac{\Gamma, x : \texttt{Lift}\ \tau \vdash e_1 : \texttt{Lift}\ \tau \quad \Gamma, x : \texttt{Lift}\ \tau \vdash e_2 : \rho}{\Gamma \vdash \texttt{letrec}\ x\texttt{:}\tau\texttt{=}e_1\ \texttt{in}\ e_2 : \rho}$$

This solution is not very satisfactory. For a start, it cannot type:

```
letrec f = \x. ... in ...
```

because the type of a lambda abstraction has the form $\tau \to \rho$, not $\texttt{Lift}\ \tau$, and lifting all functions raises the spectre of having to force the definition on each recursive call. Nor can it type recursive definitions of `ST` computations. Furthermore, this loss of expressiveness is completely unnecessary, since a function type whose result type is pointed is itself pointed; and any `ST` computation is pointed. The right solution is to fix (REC) by adding a side condition that τ must be pointed:

$$(RECb)\quad \frac{\Gamma, x : \tau \vdash e_1 : \tau \qquad \Gamma, x : \tau \vdash e_2 : \rho \qquad \vdash \tau\ \text{pointed}}{\Gamma \vdash \texttt{letrec}\ x\texttt{:}\tau\texttt{=}e_1\ \texttt{in}\ e_2 : \rho}$$

Figure 6 gives rules for determining when a type is pointed. Unfortunately, the extension to a polymorphic type system is problematic: *is the type α 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 3 and $\texttt{ret}_{\texttt{Lift}}$ 3, because one has a lifted type and one does not.

[6] There is a substantial literature on the categorical treatment of recursion (for example, Pitts [1996]), but the discussion of this section focuses on the specific setting of $\mathcal{CPO}$.

- We could allow type variables to range over all types, but prohibit recursion at a type variable. This would irritatingly reject recursive functions whose result type is a type variable, such as the function `nth` that selects the n'th element from a list.

$$\texttt{nth} : \forall\alpha.\texttt{Int -> (List } \alpha\texttt{) -> } \alpha$$

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

$$\texttt{nth} : \forall\alpha \in \texttt{Pointed}.\texttt{Int -> (List } \alpha\texttt{) -> } \alpha$$

  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:

$$\texttt{let x} : \tau = e \texttt{ in f x}$$

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 -> 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! Alas, 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 α? 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 $\texttt{let } x{:}\tau = e \texttt{ in } b$ 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 `let`s 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 7 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, $\texttt{<}e\texttt{>}$, that suspends the evaluation of e, and a new constant, `force`, that forces the suspension returned by its argument. There is one new type, $\texttt{<}\rho\texttt{>}$, which is the type of $\texttt{<}e\texttt{>}$ if e has type ρ. The two new type rules, (*DELAY*) and (*FORCE*) are just as you would expect.

Another new feature is that types are divided into *value types*, τ, and *computation types*, ρ. 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

$$\{x_1 : \tau_1, \ldots, x_n : \tau_n\} \vdash e : \rho$$

The type rules of Figure 1 apply unchanged, because we carefully used τ and ρ 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 finesses 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 `(f (read r))` is ill-typed, and hence we do not have to evaluate `(read r)` without also performing its state changes. Indeed, expressions of type $\texttt{ST } \tau$ can only occur as

$$
\begin{array}{rcl}
\textit{Computation types } \rho & ::= & M\ \tau \mid \tau \\
\textit{Value types } \tau & ::= & \texttt{Int} \mid \tau\texttt{->}\rho \mid \texttt{()} \mid \texttt{(}\tau_1\texttt{,}\tau_2\texttt{)} \\
 & \mid & \texttt{<}\rho\texttt{>} \mid \texttt{Ref}\ \tau
\end{array}
$$

$$
\begin{array}{rcl}
\textit{Terms} \quad e & ::= & x \mid k \mid e_1\ e_2 \mid \texttt{\\}x.e \mid \texttt{(}e_1\texttt{,}e_2\texttt{)} \mid \texttt{<}e\texttt{>} \\
 & \mid & \texttt{let}\ x\texttt{:}\tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \\
 & \mid & \texttt{letrec}\ x\texttt{:}\tau\ \texttt{=}\ pv\ \texttt{in}\ e \\
 & \mid & \texttt{let}_M\ x\texttt{:}\tau\ \texttt{<-}\ e_1\ \texttt{in}\ e_2 \mid \texttt{ret}_M\ e
\end{array}
$$

$$
\begin{array}{rcl}
\textit{Constants} \quad k & ::= & \ldots \mid \texttt{force} \\
\textit{Monads } M & ::= & \texttt{ST} \\
\textit{Values} \quad v & ::= & x \mid k \mid \texttt{\\}x\texttt{:}\tau\texttt{.}e \mid \texttt{<}e\texttt{>} \mid \texttt{(}v_1\texttt{,}v_2\texttt{)} \\
\textit{PValues } pv & ::= & \texttt{\\}x\texttt{:}\tau\texttt{.}e \mid \texttt{<}e\texttt{>}
\end{array}
$$

The type rules from Figure 1, plus...

$$(\mathit{DELAY}) \qquad \frac{\Gamma \vdash e : \rho}{\Gamma \vdash \texttt{<}e\texttt{>} : \texttt{<}\rho\texttt{>}}$$

$$(\mathit{FORCE}) \qquad \Gamma \vdash \texttt{force} : \texttt{<}\rho\texttt{>}\ \texttt{->}\ \rho$$

Figure 7: Extra syntax and type rules for $\mathcal{L}_2$

The translation $\mathcal{M}$ from ML to $\mathcal{L}_2$
is textually the same as in Figure 5

$$
\begin{array}{rcl}
\mathcal{H}[\![\mathbf{Int}]\!] & = & \texttt{Int} \\
\mathcal{H}[\![S * T]\!] & = & \texttt{(<}\mathcal{H}[\![S]\!]\texttt{>,<}\mathcal{H}[\![T]\!]\texttt{>)} \\
\mathcal{H}[\![()]\!] & = & () \\
\mathcal{H}[\![S \to T]\!] & = & \texttt{<}\mathcal{H}[\![S]\!]\texttt{>}\ \texttt{->}\ \mathcal{H}[\![T]\!] \\
\mathcal{H}[\![\mathbf{ST}\ T]\!] & = & \texttt{ST}\ \texttt{<}\mathcal{H}[\![T]\!]\texttt{>} \\
\mathcal{H}[\![\mathbf{Ref}\ S]\!] & = & \texttt{Ref}\ \texttt{<}\mathcal{H}[\![S]\!]\texttt{>} \\
 & & \\
\mathcal{H}[\![x]\!] & = & \texttt{force}\ x \\
\mathcal{H}[\![i]\!] & = & i \\
\mathcal{H}[\![M\ N]\!] & = & \mathcal{H}[\![M]\!]\ \texttt{<}\mathcal{H}[\![N]\!]\texttt{>} \\
\mathcal{H}[\![\lambda x\!:\!T.M]\!] & = & \texttt{\\}x\texttt{:<}\mathcal{H}[\![T]\!]\texttt{>.}\mathcal{H}[\![M]\!] \\
\mathcal{H}[\![\mathbf{let}\ x\!:\!T = M\ \mathbf{in}\ N]\!] & = & \texttt{let}\ x\texttt{:<}\mathcal{H}[\![T]\!]\texttt{>}\ \texttt{=}\ \texttt{<}\mathcal{H}[\![M]\!]\texttt{>}\ \texttt{in} \\
 & & \mathcal{H}[\![N]\!] \\
\mathcal{H}[\![\mathbf{letrec}\ x\!:\!T = M\ \mathbf{in}\ N]\!] & = & \texttt{letrec}\ x\texttt{:<}\mathcal{H}[\![T]\!]\texttt{>}\ \texttt{=}\ \texttt{<}\mathcal{H}[\![M]\!]\texttt{>} \\
 & & \texttt{in}\ \mathcal{H}[\![N]\!] \\
\mathcal{H}[\![\mathbf{fst}\ M]\!] & = & \texttt{force (fst}\ \mathcal{H}[\![M]\!]\texttt{)} \\
\mathcal{H}[\![\mathbf{pair}\ M\ N]\!] & = & \texttt{(<}\mathcal{H}[\![M]\!]\texttt{>,<}\mathcal{H}[\![N]\!]\texttt{>)} \\
\mathcal{H}[\![M + N]\!] & = & \texttt{+}\ \mathcal{H}[\![M]\!]\ \mathcal{H}[\![N]\!] \\
\mathcal{H}[\![\mathbf{wr}\ M\ N]\!] & = & \texttt{wr}\ \mathcal{H}[\![M]\!]\ \mathcal{H}[\![N]\!] \\
 & & \ldots\text{similarly } \mathbf{new}, \mathbf{rd} \\
\mathcal{H}[\![\mathbf{let_{ST}}\ x\!:\!T \leftarrow M\ \mathbf{in}\ N]\!] & = & \texttt{let}_{\texttt{ST}}\ x\texttt{:<}\mathcal{H}[\![T]\!]\texttt{>}\ \texttt{<-}\ \mathcal{H}[\![M]\!] \\
 & & \texttt{in}\ \mathcal{H}[\![N]\!] \\
\mathcal{H}[\![\mathbf{ret_{ST}}\ M]\!] & = & \texttt{ret}_{\texttt{ST}}\ \mathcal{H}[\![M]\!]
\end{array}
$$

Figure 9: Translations of "ML" and "Haskell" into $\mathcal{L}_2$

the right hand side of a $\texttt{let}_{\texttt{ST}}$, the body of a function, or as the value of the whole program. Finally, when polymorphism is introduced, type variables range over value types only.

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 "$\rightharpoonup$", 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 $\mathcal{E}[\![e_1\ e_2]\!]\rho$ says that if both $\mathcal{E}[\![e_1]\!]\rho$ and $\mathcal{E}[\![e_2]\!]\rho$ are defined then the result is just the application of those two values; otherwise there is no equation that applies for $\mathcal{E}[\![e_1\ e_2]\!]\rho$, so it too is undefined.

The `<_>` type constructor is modeled using lifting; the semantics of `force` and `<_>` move to and fro between lifted CPOs and partial functions. It may seem odd that we use two different notations — $\texttt{Lift}\ \tau$ in $\mathcal{L}_1$and $\texttt{<}\tau\texttt{>}$ 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:

$$S \rightharpoonup T \cong S \to T_\perp$$

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 `<_>` on the arguments of functions and data constructors. Concerning terms, the thunk-former `<_>` 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 `fst` or `snd`.

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

The main benefit of using $\mathcal{L}_2$ is that its semantics permit an eager 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* $\texttt{<}\tau\texttt{>}$*, or a type variable (which might be instantiated to* $\texttt{<}\tau\texttt{>}$*), 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 more or less have to restrict `letrec`s 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{array}{rcll}
\mathcal{T} : \mathit{Type} & \rightarrow & \mathcal{CPO} & \\
\mathcal{T}[\![\texttt{Int}]\!] & = & \mathcal{Z} & \\
\mathcal{T}[\![\tau_1\texttt{->}\tau_2]\!] & = & \mathcal{T}[\![\tau_1]\!] \rightharpoonup \mathcal{T}[\![\tau_2]\!] & \\
\mathcal{T}[\![\texttt{(}\tau_1\texttt{,}\tau_2\texttt{)}]\!] & = & \mathcal{T}[\![\tau_1]\!] \times \mathcal{T}[\![\tau_2]\!] & \\
\mathcal{T}[\![\texttt{<}\tau\texttt{>}]\!] & = & \tau_\perp & \\
\mathcal{T}[\![\texttt{ST}\ \tau]\!] & = & \mathbf{State} \rightharpoonup (\mathcal{T}[\![\tau]\!] \times \mathbf{State}) & \\
\mathcal{T}[\![\texttt{Ref}\ \tau]\!] & = & \mathcal{N} & \\
& & & \\
\mathcal{E} : \mathit{Term}_\tau \rightarrow \mathit{Env} & \rightharpoonup & \mathcal{T}[\![\tau]\!] & \\
\mathcal{E}[\![x]\!]\rho & = & \rho(x) & \\
\mathcal{E}[\![k]\!]\rho & = & k & \\
\mathcal{E}[\![e_1\ e_2]\!]\rho & = & (\mathcal{E}[\![e_1]\!]\rho)\ (\mathcal{E}[\![e_2]\!]\rho), & \text{if } \mathcal{E}[\![e_1]\!]\rho \text{ and } \mathcal{E}[\![e_2]\!]\rho \text{ are defined} \\
\mathcal{E}[\![\texttt{\\}x\texttt{.}e]\!]\rho & = & \lambda y.\mathcal{E}[\![e]\!]\rho[x := y] & \\
\mathcal{E}[\![\texttt{(}e_1\texttt{,}e_2\texttt{)}]\!]\rho & = & (\mathcal{E}[\![e_1]\!]\rho, \mathcal{E}[\![e_2]\!]\rho), & \text{if } \mathcal{E}[\![e_1]\!]\rho \text{ and } \mathcal{E}[\![e_2]\!]\rho \text{ are defined} \\
\mathcal{E}[\![\texttt{let}\ x\texttt{:}\tau\texttt{=}e_1\ \texttt{in}\ e_2]\!]\rho & = & \mathcal{E}[\![e_2]\!]\rho[x := \mathcal{E}[\![e_1]\!]\rho], & \text{if } \mathcal{E}[\![e_1]\!]\rho \text{ is defined} \\
\mathcal{E}[\![\texttt{letrec}\ x\texttt{:}\tau\texttt{=}pv\ \texttt{in}\ e]\!]\rho & = & \mathcal{E}[\![e]\!](\mathit{fix}(\lambda\rho'.\rho[x := \mathcal{E}[\![pv]\!]\rho'])) & \\
\mathcal{E}[\![\texttt{let}_M\ x\texttt{:}\tau\texttt{<-}e_1\ \texttt{in}\ e_2]\!]\rho & = & \mathit{bind}_M\ (\mathcal{E}[\![e_1]\!]\rho)\ (\lambda y.\mathcal{E}[\![e_2]\!]\rho[x := y]), & \text{if } \mathcal{E}[\![e_1]\!]\rho \text{ is defined} \\
\mathcal{E}[\![\texttt{ret}_M\ e]\!]\rho & = & \mathit{unit}_M\ (\mathcal{E}[\![e]\!]\rho), & \text{if } \mathcal{E}[\![e]\!]\rho \text{ is defined} \\
\mathcal{E}[\![\texttt{<}e\texttt{>}]\!]\rho & = & (\mathcal{E}[\![e]\!]\rho)_\perp, & \text{if } \mathcal{E}[\![e]\!]\rho \text{ is defined} \\
& & \perp, & \text{otherwise} \\
& & & \\
\mathit{fst}\ (a,b) & = & a & \\
\mathit{snd}\ (a,b) & = & b & \\
\mathit{force}\ a_\perp & = & a & \\
\mathit{bind}_{ST}\ m\ k\ s & = & k\ r\ s', & \text{if } m\ s = (r,s')_\perp \\
\mathit{unit}_{ST}\ m\ s & = & (m,s) & \\
\mathit{new}\ v\ s & = & (r, s[r \mapsto v]) & \text{where } r \notin \mathit{dom}(s) \\
\mathit{rd}\ r\ s & = & (s\ r, s), & \text{if } r \in \mathit{dom}(s) \\
\mathit{wr}\ r\ v\ s & = & ((), s[r \mapsto v]), & \text{if } r \in \mathit{dom}(s)
\end{array}
$$

Figure 8: Semantics of $\mathcal{L}_2$

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 `<`e`>`, where $e : \tau$, is drawn from the pointed CPO $\mathcal{T}[\![\tau]\!]_\perp$.

Fortunately, the syntactic restriction of `letrec` does not lose any useful expressiveness. ML insists that `letrec`s 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 $\mathcal{L}_2$.

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 $\texttt{let}_{\texttt{ST}}$, 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 $\texttt{let}_{\texttt{ST}}\ x\ \texttt{<-}\ e\ \texttt{in}\ b$ 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 $\texttt{let}\ x\,\texttt{=}\,e\ \texttt{in}\ b$.

However, if we have just one form of `let` we lose valuable optimising transformations. In particular, the sequence of computations in `ST` must be maintained, whereas `let` bindings can be re-ordered 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 $\mathcal{L}_1$). To take a simple example, the following transformation is not in general valid for $\texttt{let}_{\texttt{ST}}$, but is valid for vanilla `let` (assuming there are no name clashes):

$$
\begin{array}{c}
\texttt{let}\ x_1\,\texttt{=}\,e_1\ \texttt{in}\ \texttt{let}\ x_2\,\texttt{=}\,e_2\ \texttt{in}\ b \\
= \\
\texttt{let}\ x_2\,\texttt{=}\,e_2\ \texttt{in}\ \texttt{let}\ x_1\,\texttt{=}\,e_1\ \texttt{in}\ b
\end{array}
$$

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 $\mathcal{L}_1$ vs $\mathcal{L}_2$

What have we lost in the transition from $\mathcal{L}_1$ to $\mathcal{L}_2$, apart from a somewhat more complicated semantics? One loss is $\mathcal{L}_1$'s ability to describe types whose values are sure to terminate. If a $\mathcal{L}_1$ function has type `Int->Int` then a call to the function cannot diverge; but the same is not true of $\mathcal{L}_2$. This does not have much impact on a compiler, but it make programmer reasoning about $\mathcal{L}_2$ programs more complicated.

Another important difference is that $\mathcal{L}_2$ has a weaker β rule. $\mathcal{L}_1$ has full β-conversion. That is, for any expressions e and b:

$$\texttt{let}\ x = e\ \texttt{in}\ b = b[e/x]$$

(A similar rule holds for application, of course.) In $\mathcal{L}_2$, however, β does not hold in general. A particular case of this is that if x is not mentioned in b then in $\mathcal{L}_1$ the binding can be discarded; in $\mathcal{L}_2$ the binding can only be discarded if the right-hand side is a value.

However β_V — a restricted version form of β that allows only *values* to be substituted — is valid in $\mathcal{L}_2$. Values are defined in Figure 7, and include variables, constants, and lambda abstractions, as usual. However, values also include thunks. Hence *any Haskell β reduction has a corresponding β_V reduction in its $\mathcal{L}_2$ translation.* Thus, the restriction to β_V will not prevent a Haskell compiler from doing anything it can do in an implicitly lazy language with a full β 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. [1995] describes, we can give a call-by-need semantics to $\mathcal{L}_1$ by weakening β to β_V and adding a garbage-collection rule that allows an unused `let` binding to be discarded. An analogous result holds in $\mathcal{L}_2$: we can obtain call-by-need semantics by replacing `<`e`>` by `<`v`>` in the definition of values in Figure 7.

5.2 $\mathcal{L}_2$ 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 $\mathcal{L}_2$ is attractive in its own right to either community in isolation, because one might get better code from an $\mathcal{L}_2$-based compiler.

For the Haskell compiler writer $\mathcal{L}_2$ 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 $\mathcal{L}_2$ 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 $\mathcal{L}_2$, however, local transformations can perform, and record the results of, a simple incremental effects analysis. For example, consider the following ML function:

```
fun f x = fst (fst x)
```

If we translate this into $\mathcal{L}_2$ we obtain:

```
f = retST (λx. letST a2 <- letST a1 <- retST x in
                          retST (fst a1)
               in
               retST (fst a2))
```

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

```
f = retST (λx. let a1 = x in
               let a2 = fst a1 in
               retST (fst a2))
```

Now the $\texttt{ret}_{\texttt{ST}}$ can be floated outwards, to give:

```
f = retST (λx.retST (fst (fst x)))
```

In this form, the inner $\texttt{ret}_{\texttt{ST}}$ 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 $\texttt{ret}_{\texttt{ST}}$ may cancel with $\texttt{let}_{\texttt{ST}}$ 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*, `fw`, thus:

```
f  =retST (λx.retST (fw x))
fw =λx.(... body of f ...)
```

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 [1983]; Wadler [1989]). For example, in the pure polymorphic lambda calculus, a function f with type $\forall\alpha.\alpha \to \alpha \to \alpha$ satisfies the theorem:

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

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 $\mathit{fix} : \forall\alpha.(\alpha \to \alpha) \to \alpha$ forces the restriction that the h functions be strict (map $\perp$ to $\perp$) and inductive (i.e. continuous). This is the situation in Haskell, for example.

Adding polymorphic sequencing, say through an operator $\mathit{seq} : \forall\alpha, \beta.\alpha \to \beta \to \beta$ 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 $\mathcal{L}_1$ and $\mathcal{L}_2$? 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 $\mathcal{L}_2$, 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 $\mathcal{L}_1$ 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 $\mathcal{L}_1$ and $\mathcal{L}_2$ is to do with forcing evaluation. $\mathcal{L}_1$ has no polymorphic forcing operation, so has no consequent weakening of its parametricity property. $\mathcal{L}_2$ does, however — it is built into its eager function application. Thus for $\mathcal{L}_2$ the parametricity theorem demands the h functions to be everywhere defined.

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

$$\begin{array}{l}\forall A, A', B, B' \; . \; \forall h_1 : A \to A', h_2 : B \to B' \; . \; \forall x : A, y : B \; . \\ \quad h_1 \; (K \; x \; y) = K \; (h_1 \; x) \; (h_2 \; y)\end{array}$$

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 $\mathcal{L}_2$'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 `r` with type $\forall\alpha.$`Ref` α then we would be able to write the following erroneous code:

```
let_ST _:()<-wr (r Int) 2 in
let_ST f:(Int->Int)<-rd (r (Int->Int)) in
ret_ST (f 3)
```

However in both $\mathcal{L}_1$ and $\mathcal{L}_2$ any expression of type $\forall\alpha.$`Ref` α is undefined in any environment! The only way to construct a value of `Ref` type is with `new`, which returns a value of type `ST (Ref` τ`)`. The only way to strip off the `ST` constructor is with $\texttt{let}_{\texttt{ST}}$. Looking at the typing rule for $\texttt{let}_{\texttt{ST}}$, we can see that bound variable must have type `Ref` τ.

SML's so-called "value restriction" conservatively restricts generalisation in `let` bindings precisely to avoid the construction of such polymorphic references. We conjecture (though we have not proved) that $\mathcal{L}_1$ and $\mathcal{L}_2$ 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 $\mathcal{L}_2$. 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 "`$`", thus:

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

Here, assuming `(f y)` has type `int`, `x` is bound to a thunk of type `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 $\mathcal{L}_2$'s thunks, but that turned out not to be the case. The semantics of ML thunks is considerably more complicated than that of $\mathcal{L}_2$'s thunks, because of the interaction with state. Consider the following ML expression:

```
let val rec x = $(let val y = !r - 1 in
                  r:=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 `$`e by $\lambda().e$.) Furthermore, these multiple activations can each have a different value, because they each read the state.

$\mathcal{L}_2$'s thunks have a much simpler semantics. A thunk has only one value, and there can be at most one activation of the thunk[7]. The key insight is that *evaluation of a* $\mathcal{L}_2$ *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_ST v : Int <- rd r in wr (v+1)> in e
```

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

```
let_ST a1 : () <- force x in let_ST a2 : () <- force x in ...
```

What this means, though, is that the more complicated semantics of ML thunks have to be expressed explicitly in $\mathcal{L}_2$, presumably by coding them up using explicit references.

[7] 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 other's 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 $\mathcal{L}_2$; 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 $\mathcal{L}_1$'s ability to describe certainly-terminating computations with $\mathcal{L}_2$'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 $\mathcal{L}_2$ (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

Z Ariola, M Felleisen, J Maraist, M Odersky & P Wadler [1995], "A call by need lambda calculus," in *22nd ACM Symposium on Principles of Programming Languages, San Francisco*, ACM, Jan 1995, 233–246.

N Benton & PL Wadler [1996], "Linear logic, monads and the lambda calculus," in *Proceedings of the 11th IEEE Symposium on Logic in Computer Science, Brunswick, New Jersey*, IEEE Press, July 1996.

M Blume & AW Appel [1997], "Lambda-splitting: a higher-order approach to cross-module optimization," in *Proc International Conference on Functional Programming, Amsterdam*, ACM, June 1997, 112–124.

R Cockett & T Fukushima [1992], "About Charity," TR 92/480/18, Department of Computer Science, University of Calgary, June 1992.

DP Friedman & DS Wise [1976], "CONS should not evaluate its arguments," *Automata, Languages, and Programming*, July 1976, 257–281.

A Gill, J Launchbury & SL Peyton Jones [1993], "A short cut to deforestation," in *Proc Functional Programming Languages and Computer Architecture, Copenhagen*, ACM, June 1993, 223–232.

J-Y Girard [1990], "The System F of variable types: fifteen years later," in *Logical Foundations of Functional Programming*, G Huet, ed., Addison-Wesley, 1990.

T Hagino [1987], "A Categorical Programming Language," PhD thesis, Department of Computer Science, University of Edinburgh, 1987.

R Harper & JC Mitchell [1993], "On the type structure of Standard ML," *ACM Transactions on Programming Languages and Systems* 15(2), April 1993, 211–252.

R Harper & G Morrisett [1995], "Compiling polymorphism using intensional type analysis," in *22nd ACM Symposium on Principles of Programming Languages, San Francisco*, ACM, Jan 1995, 130–141.

R Harper & C Stone [1997], "A type-theoretic semantics for Standard ML 1996," Department of Computer Science, Carnegie Mellon University, 1997.

BT Howard [1996], "Inductive, co-inductive, and pointed types," in *Proc International Conference on Functional Programming, Philadelphia*, ACM, May 1996.

MP Jones [1994], "Dictionary-free overloading by partial evaluation," in *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), Orlando, Florida*, ACM, June 1994.

MP Jones [1996], "Using parameterized signatures to express modular structure," in *23rd ACM Symposium on Principles of Programming Languages, St Petersburg Beach, Florida*, ACM, Jan 1996, 68–78.

R Jones [1992], "Tail recursion without space leaks," *Journal of Functional Programming* 2(1), Jan 1992, 73–80.

J Launchbury & G Baraki [1996], "Representing Demand by Partial Projections," *Journal of Functional Programming* 6(4), 1996.

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.

J Launchbury & SL Peyton Jones [1995], "State in Haskell," *Lisp and Symbolic Computation* 8(4), Dec 1995, 293–342.

X Leroy [1992], "Unboxed objects and polymorphic typing," in *19th ACM Symposium on Principles of Programming Languages, Albuquerque*, ACM, Jan 1992.

R Milner & M Tofte [1990], *The definition of Standard ML*, MIT Press, 1990.

JC Mitchell & AR Meyer [1985], "Second-order logical relations," in *Logics of Programs*, R Parikh, ed., Springer Verlag LNCS 193, 1985.

E Moggi [1991], "Notions of computation and monads," *Information and Computation* 93, 1991, 55–92.

C Okasaki [1996], "Purely functional data structures," PhD thesis, CMU-CS-96-177, Department of Computer Science, Carnegie Mellon University, Sept 1996.

J Peterson, K Hammond, L Augustsson, B Boutel, W Burton, J Fasel, AD Gordon, RJM Hughes, P Hudak, T Johnsson, MP Jones, E Meijer, SL Peyton Jones, A Reid & PL Wadler [1997], "Haskell 1.4: a non-strict, purely functional language," Available at `http://-haskell.org`, April 1997.

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.

SL Peyton Jones, AJ Gordon & SO Finne [1996], "Concurrent Haskell," in *23rd ACM Symposium on Principles of Programming Languages, St Petersburg Beach, Florida*, ACM, Jan 1996, 295–308.

SL Peyton Jones & J Launchbury [1991], "Unboxed values as first class citizens," in *Functional Programming Languages and Computer Architecture (FPCA'91), Boston*, Hughes, ed., LNCS 523, Springer Verlag, Sept 1991, 636–666.

SL Peyton Jones & E Meijer [1997], "Henk: a typed intermediate language," in *ACM Workshop on Types in Compilation, Amsterdam*, R Harper & R Muller, eds., June 1997.

SL Peyton Jones, WD Partain & A Santos [1996], "Let-floating: moving bindings to give faster programs," in *Proc International Conference on Functional Programming, Philadelphia*, ACM, May 1996.

SL Peyton Jones & PL Wadler [1993], "Imperative functional programming," in *20th ACM Symposium on Principles of Programming Languages (POPL'93), Charleston*, ACM, Jan 1993, 71–84.

AM Pitts [1996], "Relational properties of domains," *Information and Computation* 127, 1996, 66–90.

JC Reynolds [1983], "Types, abstraction and parametric polymorphism," in *Information Processing 83*, REA Mason, ed., North-Holland, 1983, 513–523.

A Sabry [1997], "What is a purely functional language," *Journal of Functional Programming* (to appear), 1997.

A Sabry & PL Wadler [1996], "A reflection on call-by-value," in *Proc International Conference on Functional Programming, Philadelphia*, ACM, May 1996, 13–24.

Z Shao [1997a], "Flexible representation analysis," in *Proc International Conference on Functional Programming, Amsterdam*, ACM, June 1997.

Z Shao [1997b], "An Overview of the FLINT/ML compiler," in *ACM Workshop on Types in Compilation, Amsterdam*, R Harper & R Muller, eds., June 1997.

Z Shao & AW Appel [1995], "A type-based compiler for Standard ML," in *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'95), La Jolla*, ACM, June 1995, 116–129.

D Tarditi, G Morrisett, P Cheng, C Stone, R Harper & P Lee [1996], "TIL: A Type-Directed Optimizing Compiler for ML," in *SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'96), Philadelphia*, ACM, May 1996.

M Tofte [1990], "Type inference for polymorphic references," *Information and Computation* 89(1), Nov 1990.

A Tolmach & D Oliva [1997], "From ML to Ada(!?!)," Department of Computer Science and Engineering, Oregon Graduate Institute (and submitted to Journal of Functional Programming), 1997.

DA Turner [1995], "Elementary strong functional programming," in *Functional Programming Languages in Education, Nigmegen*, Springer Verlag LNCS 1022, Dec 1995.

PL Wadler [1989], "Theorems for free!," in *Fourth International Conference on Functional Programming and Computer Architecture, London*, MacQueen, ed., Addison Wesley, 1989.

PL Wadler [1992a], "Comprehending monads," *Mathematical Structures in Computer Science* 2, 1992, 461–493.

PL Wadler [1992b], "The essence of functional programming," in *19th ACM Symposium on Principles of Programming Languages, Albuquerque*, ACM, Jan 1992, 1–14.