Haskell Is Not Not ML

Ben Rudiak-Gould[1], Alan Mycroft[1], and Simon Peyton Jones[2]

[1] University of Cambridge Computer Laboratory
[2] Microsoft Research Cambridge
{br276, am}@cl.cam.ac.uk, simonpj@microsoft.com

Abstract. We present a typed calculus IL ("intermediate language") which supports the embedding of ML-like (strict, eager) and Haskell-like (non-strict, lazy) languages, without favoring either. IL's type system includes negation (continuations), but not implication (function arrow). Within IL we find that lifted sums and products can be represented as the double negation of their unlifted counterparts. We exhibit a compilation function from IL to AM—an abstract von Neumann machine—which maps values of ordinary and doubly negated types to heap structures resembling those found in practical implementations of languages in the ML and Haskell families. Finally, we show that a small variation in the design of AM allows us to treat any ML value as a Haskell value at runtime without cost, and project a Haskell value onto an ML type with only the cost of a Haskell `deepSeq`. This suggests that IL and AM may be useful as a compilation and execution model for a new language which combines the best features of strict and non-strict functional programming.

1 Introduction

Every functional language in use today is either strict and eagerly evaluated or non-strict and lazily evaluated. Though most languages make some provision for both evaluation strategies, it is always clear which side their bread is buttered on: one evaluation strategy is automatic and transparent, while the other requires micromanagement by the programmer.

The dichotomy is surprising when one considers how similar lazy functional programs and eager functional programs look in practice. Most of the differences between SML and Haskell are independent of evaluation order (syntax, extensible records, module systems, type classes, monadic effect system, rank-2 types...). Were it not for those differences, it would in many cases be difficult to tell which language a given code fragment was actually written in. Why, then, is there no *hybrid language* which can understand code like

```
foldl f z l = case l of []     -> z
                        (x:xs) -> foldl f (f z x) xs
```

in some way that abstracts over possible evaluation orders?

Designing such a language turns out to be quite difficult. Reducing the significant notational and cognitive burdens of mixed strict/non-strict programming

P. Sestoft (Ed.): ESOP 2006, LNCS 3924, pp. 38–53, 2006.

is an open research problem, which we do not attempt to solve in this paper. Instead we address a prerequisite for the success of any hybrid language, which is the possibility of implementing it easily and efficiently enough to be competitive with dedicated strict and lazy languages. That is, we discuss the optimization and back end phases of a hybrid language compiler, leaving the front end for future work.

In Section 2 we introduce a compiler intermediate language ("IL") which can support conventional strict and non-strict source languages through different front-end translations. IL is the centerpiece and the main contribution of this paper. Section 3 presents toy strict and non-strict languages ("SL" and "LL") and their translations to IL. SL and LL are entirely conventional in design, and are included only as examples of translations to IL. Section 4 introduces an abstract machine ("AM") with a von Neumann architecture, which can act as an execution environment for IL. Our goal is that compilation of SL or LL via IL to AM should be competitive (in code size and execution speed) with compilation of SL or LL to purpose-designed, incompatible abstract machines.

With this architecture we can compile ML-like and Haskell-like source code to a common abstract machine with a single heap. The strict and non-strict languages cannot share data structures, since they have incompatible type systems, but they can exchange data via appropriate marshalling code (written in IL). In Section 5 we aim to get rid of the marshalling and enable direct sharing. By carefully choosing the representation of the heap data, we can arrange that the natural injection from ML values into Haskell values is a no-op at runtime, and the natural projection from Haskell values to pointed ML values carries only the cost of a Haskell `deepSeq` operation (in particular, it need not perform copying).

Space constraints have forced us to omit or gloss over many interesting features of IL in this conference paper. Interested readers may find additional material at the project website [7].

1.1 A Note on Types

We handle recursive types with a form $\nu\alpha.\, T$, which denotes the type T with free occurrences of α replaced by T itself. The ν form can be seen as extending the tree of types to a graph with cycles; $\nu\alpha$ simply gives a name to the graph node which it annotates. For example, $\nu\alpha.\, \neg\alpha$ describes a single negation node pointing to itself, while $\nu\alpha.\, \alpha$ is meaningless, since α refers to no node. In this paper we will treat the ν form as a metasyntactic description of a type graph, rather than a feature of the concrete syntax of types. This allows us to omit rules for syntactic manipulation of ν forms, which would complicate the presentation and yield no new insight.

Rather than attempt to distinguish between inductive types (à la ML) and coinductive types (à la Haskell), we make all types coinductive, with the caveat that types may contain values which have no representation as terms in the language. This is already true of recursive datatypes in Haskell (for example, uncountably many values inhabit the Haskell list type `[Bool]`, but there are only countably many expressions of type `[Bool]`) and it is true of functions in both ML and Haskell (if S is infinite then $S \to$ `Bool` is uncountable).

Deciding whether to include parametric polymorphism was difficult. There are several cases in which we would like to speak of polymorphic types, and one case in which we must do so; on the other hand the introduction of polymorphism into our languages has no novel features, and would in most cases simply clutter the discussion and the figures. Instead we adopt a compromise approach. Within IL, SL and LL, type variables (except those quantified by ν) simply represent unknown types; there is no instantiation rule and no polymorphic quantification. We permit ourselves, however, to speak of instantiation within the text of the paper. E.g. we may say that a term E has the "type" $\forall\alpha.\,(\alpha\,\&\,\alpha)$, by which we mean that for any type T, $E[T/\alpha]$ is well-typed and has the type $(T\,\&\,T)$.

2 IL

IL is a continuation-passing calculus with a straightforward syntax and semantics. It makes no explicit mention of strictness or non-strictness, but it contains both notions in a way which will be explained in Section 2.3.

IL types are shown in Fig. 1, and IL expressions in Fig. 2. We will use the words "expression" and "term" interchangeably in this paper. The type $\mathbf{0}$ has a special role in IL; the distinction between $\mathbf{0}$ and non-$\mathbf{0}$ types, and associated terms and values, will recur throughout this paper.[1] Therefore we adopt the convention that T ranges only over non-$\mathbf{0}$ types (mnemonic True), while U ranges over all types. For expressions, E ranges only over those of non-$\mathbf{0}$ type; F ranges over those of type $\mathbf{0}$ (mnemonic: False); and G ranges over all expressions (General). Note in particular that Figs. 1 and 2 imply that we do not permit types such as $\neg\mathbf{0}$ and $(\mathbf{0} \vee \mathbf{1})$: the only type containing $\mathbf{0}$ is $\mathbf{0}$ itself.

In the spirit of Martin-Löf type theory and the Curry-Howard isomorphism, we give both a logical and an operational interpretation to IL types and expressions. Logically, types are formulas, and expressions are proofs of those formulas; operationally, types are sets of values and expressions evaluate to a value from the set.

Syntax	Logical meaning	Operational meaning
$\mathbf{0}$	Contradiction.	The empty type.
$\mathbf{1}$	Tautology.	The unit type.
$T_1 \,\&\, T_2$	Conjunction (there are proofs of T_1 and T_2).	Unlifted product.
$T_1 \vee T_2$	Disjunction (there is a proof of T_1 or of T_2).	Unlifted sum.
$\neg T$	From T it is possible to argue a contradiction.	Continuation (see text).
α, β, γ	Free type variables (of non-$\mathbf{0}$ type).	

Fig. 1. Syntax and gloss of IL types

[1] Our type $\mathbf{0}$ is conventionally called $\bot$, but in this paper we use the symbol $\bot$ for another purpose.

Syntax	Logical meaning	Operational meaning
x, y, k	Free variables.	
$\lambdabar(x{:}T).\,F$	A *reductio ad absurdum* proof of $\neg T$.	Builds a closure.
$E_1 \bowtie E_2$	Proves $\mathbf{0}$ (a contradiction) from E_1, of type $\neg T$, and E_2, of type T.	Enters a closure.
$()$	Proves $\mathbf{1}$.	Unit constructor.
(E_1, E_2)	Proves a conjunction by proving its conjuncts.	Pair constructor.
inl E	Proves a disjunction by its left case.	Union constructor.
inr E	Proves a disjunction by its right case.	Union constructor.
fst E	Proves T_1, where $E : (T_1 \,\&\, T_2)$.	Pair deconstructor.
snd E	Proves T_2, where $E : (T_1 \,\&\, T_2)$.	Pair deconstructor.
case ...	**case** $E\,\{(x)G_1; (y)G_2\}$ is a case analysis of a disjunction.	Union deconstructor.

Fig. 2. Syntax and gloss of IL expressions

A striking feature of IL, when compared with most programming calculi, is that its type system includes logical negation, but no implication (function arrow). Operationally, $\neg T$ is a *continuation* which takes a value of type T and never returns. Logically, the only way to prove $\neg T$ is by reductio ad absurdum: to prove $\neg T$, we show that any proof of T can be used to construct a proof of $\mathbf{0}$. We introduce continuations with the symbol $\lambdabar$ and eliminate them with infix $\bowtie$, reserving λ and juxtaposition for function types (used later in SL and LL).

In its use of $\neg$ rather than a function arrow, IL resembles Wadler's dual calculus [9]. IL, however, has only terms (no coterms), and is intuitionistic (not classical).

To simplify the discussion, we will often omit type signatures when they are uninteresting, and we will allow tuple-matching anywhere a variable is bound, so for example $\lambdabar(x, y).\ \cdots x \cdots y \cdots$ is short for $\lambdabar w.\ \cdots \mathbf{fst}\ w \cdots \mathbf{snd}\ w \cdots$.

2.1 Typing and Operational Semantics of IL

Fig. 3 lists the typing rules for IL expressions, and Fig. 4 gives a small-step operational semantics for IL, with the reduction relation $\rightsquigarrow$. These satisfy the subject reduction property that if $\Gamma \vdash G : U$ and $G \rightsquigarrow G'$, then $\Gamma \vdash G' : U$. The proof is straightforward.

We say that a term is a *value* if it has no free variables and is not subject to any reduction rules. The values in IL are

$$V ::= () \;\mid\; (V, V) \;\mid\; \mathbf{inl}\ V \;\mid\; \mathbf{inr}\ V \;\mid\; \lambdabar(x{:}T).\,F$$

Note that there are no values of type $\mathbf{0}$.

2.2 Nontermination in IL

There is no **fix** or **letrec** form in IL, but we can construct nonterminating expressions even without it. For example, let E be $(\lambdabar(x : \nu\alpha.\,\neg\alpha).\,x \bowtie x)$; then

$$
\frac{}{\Gamma,(x{:}T) \vdash x : T}
\qquad
\frac{\Gamma,(x{:}T) \vdash F : \mathbf{0}}{\Gamma \vdash \lambda(x{:}T).\,F : \neg T}
\qquad
\frac{\Gamma \vdash E_1 : \neg T \qquad \Gamma \vdash E_2 : T}{\Gamma \vdash E_1 \bowtie E_2 : \mathbf{0}}
$$

$$
\frac{\Gamma \vdash E_1 : T_1 \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash (E_1, E_2) : (T_1 \,\&\, T_2)}
\qquad
\frac{\Gamma \vdash E : (T_1 \,\&\, T_2)}{\Gamma \vdash \mathbf{fst}\ E : T_1}
\qquad
\frac{\Gamma \vdash E : (T_1 \,\&\, T_2)}{\Gamma \vdash \mathbf{snd}\ E : T_2}
$$

$$
\frac{\Gamma \vdash E : T_1}{\Gamma \vdash \mathbf{inl}\ E : (T_1 \vee T_2)}
\qquad
\frac{\Gamma \vdash E : T_2}{\Gamma \vdash \mathbf{inr}\ E : (T_1 \vee T_2)}
\qquad
\frac{}{\Gamma \vdash () : \mathbf{1}}
$$

$$
\frac{\Gamma \vdash E : (T_1 \vee T_2) \qquad \Gamma,(x{:}T_1) \vdash G_1 : U \qquad \Gamma,(y{:}T_2) \vdash G_2 : U}{\Gamma \vdash \mathbf{case}\, E\, \{(x)G_1; (y)G_2\} : U}
$$

Fig. 3. Well-typed IL expressions

$$
\begin{array}{rcl|l}
\mathbf{fst}\ (E_1, E_2) & \rightsquigarrow & E_1 & \text{Evaluation context} \\
\mathbf{snd}\ (E_1, E_2) & \rightsquigarrow & E_2 & C[\,] ::= [\,] \\
\mathbf{case}\,(\mathbf{inl}\ E)\,\{(x)G_1; (y)G_2\} & \rightsquigarrow & G_1[E/x] & \quad \mid C[\,] \bowtie E \quad \mid E \bowtie C[\,] \\
\mathbf{case}\,(\mathbf{inr}\ E)\,\{(x)G_1; (y)G_2\} & \rightsquigarrow & G_2[E/y] & \quad \mid (C[\,], E) \quad \mid (E, C[\,]) \\
(\lambda x.\,F) \bowtie E & \rightsquigarrow & F[E/x] & \quad \mid \mathbf{inl}\ C[\,] \quad \mid \mathbf{inr}\ C[\,] \\
 & & & \quad \mid \mathbf{fst}\ C[\,] \quad \mid \mathbf{snd}\ C[\,] \\
\multicolumn{3}{c|}{\dfrac{G_1 \rightsquigarrow G_2}{C[G_1] \rightsquigarrow C[G_2]}} & \quad \mid \mathbf{case}\, C[\,]\,\{(x)G_1; (y)G_2\}
\end{array}
$$

Fig. 4. Operational semantics of IL

$E \bowtie E$ is well-typed, and proves $\mathbf{0}$. We will refer to this term by the name **diverge**. It is analogous to $(\lambda x.\, x\, x)(\lambda x.\, x\, x)$ in the untyped lambda calculus. It is well-typed in IL because we permit recursive types via the ν construct. In its logical interpretation, this proof is known variously as Curry's paradox or Löb's paradox; exorcising it from a formal system is not easy. In IL, we do not try to exorcise it but rather welcome its presence, since any logically consistent language would not be Turing-complete.

But in IL, unlike ML and Haskell, nontermination cannot be used to inhabit arbitrary types. The type systems of ML and Haskell, interpreted logically, are inconsistent in the classical sense of triviality: the expression (**letrec** $x() = x()$ **in** $x()$) can appear anywhere in a program, and can be given any type whatsoever.[2] IL is not trivial; rather, it is what is known as *paraconsistent.* More precisely, IL has the following properties:

– *Confluence*: for all expressions G_1, G_2, G_3, if $G_1 \overset{*}{\rightsquigarrow} G_2$ and $G_1 \overset{*}{\rightsquigarrow} G_3$, then there is a G_4 such that $G_2 \overset{*}{\rightsquigarrow} G_4$ and $G_3 \overset{*}{\rightsquigarrow} G_4$.

[2] A similar expression can be constructed without **letrec** by using an auxiliary recursive type, as in IL.

– *Strong normalization*: for any expression E (of non-$\mathbf{0}$ type) there is an integer n such that any sequence of reductions from E has length at most n.

Together these properties imply that any IL expression of non-$\mathbf{0}$ type reduces in finitely many steps to a value which does not depend on the order of reduction. In contrast, we see that *no* IL expression of type $\mathbf{0}$ reduces to a value, since there are no values of type $\mathbf{0}$. If evaluation terminates it can only be on a non-value, such as $x \bowtie E$.

2.3 Lifted and Pointed Types in IL

For any type T, there is a natural map from the expressions (including values) of type T into the values of type $\neg\neg T$: we construct a continuation of type $\neg\neg T$ which, given a continuation of type $\neg T$, passes the value of type T to it. Symbolically, we take E to $(\lambda k.\, k \bowtie E)$. We will call this mapping **lift**.

Similarly, for any type T, there is a natural map from the values of type $\neg\neg\neg T$ onto the values of type $\neg T$: we construct a continuation of type $\neg T$ which takes a value of type T, converts it to a value of type $\neg\neg T$ by **lift**, and passes that to the continuation of type $\neg\neg\neg T$. Symbolically, we take E to $(\lambda x.\, E \bowtie (\lambda k.\, k \bowtie x))$. We will call this mapping **colift**. It is easy to see that **colift** $\circ$ **lift** is the identity on types $\neg T$, so **lift** is an injection and **colift** is a surjection.

There is, however, no natural map from $\neg\neg T$ to T when T does not begin with $\neg$ (i.e. when T is not of the form $\neg T'$). In particular, $(\lambda(x : \neg T).\, \mathbf{diverge})$ of type $\neg\neg T$ has no counterpart in T, if T has no leading $\neg$.

If we say that types T_1 and T_2 are *semantically equivalent* if there exist (co)lifting maps from T_1 to T_2 and from T_2 to T_1, then the types $\neg^k T$ for T without a leading $\neg$ fall into the three semantic equivalence classes shown below, where we write $\neg^k T$ for k successive negations of T.

1. $\neg^0 T$
2. $\neg^1 T, \neg^3 T, \neg^5 T, \neg^7 T, \ldots$
3. $\neg^2 T, \neg^4 T, \neg^6 T, \neg^8 T, \ldots$

There is a natural identification of classes 1 and 3 with the types of SL and LL *values*, respectively, and of class 2 with *evaluation contexts* in both SL and LL—more precisely, the types in class 2 are the types of the continuations which receive the computed values. We will motivate this identification informally with some examples; later, explicit translations from SL/LL to IL will make it precise.

We have already observed that **lift** is an injection: what values in $\neg\neg T$ does it miss? A value of type $\neg\neg T$ is called with a continuation of type $\neg T$. It may, perhaps after some computation, pass a result V of type T to the continuation; because of IL's purity, any such value is indistinguishable from **lift** V. But there are also values of type $\neg\neg T$ which do not call the continuation at all. In principle there may be many values in this category—one could imagine aborting execution with a variety of error messages, or transferring control to an exception-handling continuation—but following tradition we will lump all of these together as a single value $\perp_{\neg\neg T}$. Clearly $\perp$ exists not just in types $\neg\neg T$ but in any type $\neg T$: consider $ƛ(x : T).\ \mathbf{diverge}$. Types of the form $\neg T$ are *pointed*, and types of the form $\neg\neg T$ are *lifted*. This applies even if T itself begins with $\neg$, so each successive double negation adds a level of lifting: e.g. the type $\neg\neg\neg\neg\mathbf{1}$ contains distinguishable values **lift lift** (), **lift** $\bot_{\neg\neg\mathbf{1}}$, and $\bot_{\neg\neg\neg\neg\mathbf{1}}$.

We have likewise observed that **colift** is a surjection, and by a similar argument we can show that it merges the two outermost levels of lifting: in the case of $\neg\neg\neg\neg\mathbf{1}$ it maps **lift lift** () to **lift** () and maps both **lift** $\bot_{\neg\neg\mathbf{1}}$ and $\bot_{\neg\neg\neg\neg\mathbf{1}}$ to the single value $\bot_{\neg\neg\mathbf{1}}$.

The maps **lift** and **colift** resemble the **unit** and **join** operations of a lifting monad, except that **colift** is slightly more general than **join**. In a lifting monad, **unit** would map from T to $\neg\neg T$ and **join** from $\neg\neg\neg\neg T$ to $\neg\neg T$.

Our discussion above overlooks the possibility that a continuation of type $\neg T$ might be called *more* than once. Multiple calls passing the same value are indistinguishable from a single call because of purity, but there are interesting terms that pass two or more distinguishable values to their continuation. An example is $ƛk.\ k \bowtie \mathbf{inr}\ (ƛx.\ k \bowtie \mathbf{inl}\ x)$ of type $\forall\alpha.\ \neg\neg(\alpha \vee \neg\alpha)$, which is an IL interpretation of the story of the devil's offer from [9]. Inasmuch as we intend to use double negation to model Haskell lifting, we would like to forbid such values. We do not discuss this problem further in this paper.

3 SL and LL

SL and LL are simple strict and non-strict programming languages which have the same syntax, but different translations to IL. They are pure functional languages; side effects are assumed to be handled via monads or some comparable approach. SL has no provision for lazy evaluation, and LL has no provision for eager evaluation.

Fig. 5 shows the syntax of SL/LL expressions (ranged over by e) and types (ranged over by t). The typing rules and operational semantics are standard, and we will not give them here. We use **case** rather than **fst** and **snd** to deconstruct pairs because it simplifies the translation slightly. The term **error** stands for a generic divergent expression like `1/0` or Haskell's `undefined`.

x, y, z		Free variables
	α, β	Free type vars.
$\lambda(x : t).\ e$ $e\ e$	$t \to t$	Functions
$()$	$\mathbf{1}$	Unit
(e, e) $\mathbf{case}\, e\ \{(x, y)e\}$	$t \otimes t$	Pairs
$\mathbf{inl}\ e$, $\mathbf{inr}\ e$ $\mathbf{case}\, e\ \{(x)e; (y)e\}$	$t \oplus t$	Unions
error		Proves any type

Fig. 5. SL/LL expressions and types

SL/LL type	SL to IL	LL to IL
$\mathcal{D}_v [\![t]\!]$	$\mathcal{D}_b [\![t]\!]$	$\neg\neg\mathcal{D}_b [\![t]\!]$
$\mathcal{D}_k [\![t]\!]$	$\neg\mathcal{D}_b [\![t]\!]$	
$\mathcal{D}_b [\![\mathbf{1}]\!]$	$\mathbf{1}$	
$\mathcal{D}_b [\![t_1 \otimes t_2]\!]$	$\mathcal{D}_v [\![t_1]\!]\ \&\ \mathcal{D}_v [\![t_2]\!]$	
$\mathcal{D}_b [\![t_1 \oplus t_2]\!]$	$\mathcal{D}_v [\![t_1]\!] \vee \mathcal{D}_v [\![t_2]\!]$	
$\mathcal{D}_b [\![t_1 \to t_2]\!]$	$\neg(\mathcal{D}_v [\![t_1]\!]\ \&\ \mathcal{D}_k [\![t_2]\!])$	
$\mathcal{D}_b [\![\alpha]\!]$	α	

Fig. 6. Translation of SL/LL to IL types

3.1 Translation to IL

Translation of SL/LL types to IL types is shown in Fig. 6. In order to model the distinction between values and expression contexts mentioned in Section 2.3 we use three different type translations, written $\mathcal{D}_b \llbracket t \rrbracket$, $\mathcal{D}_v \llbracket t \rrbracket$, and $\mathcal{D}_k \llbracket t \rrbracket$. $\mathcal{D}_v \llbracket t \rrbracket$ is the type of *v*alues on the heap (and of bindings to *v*ariables). $\mathcal{D}_k \llbracket t \rrbracket$ is the type of a *k*ontinuation which receives the result of evaluating an expression of type t. $\mathcal{D}_b \llbracket t \rrbracket$ is a "bare" type which has not yet been converted to a value or continuation type by suitable negation.

In the interest of simplicity, SL and LL support only anonymous sums and products; there is no provision for declaring new datatypes. It is worth noting that LL's type system is consequently not quite expressive enough to represent many Haskell datatypes, because Haskell does not lift at every opportunity. For example, Haskell's `Bool` type is isomorphic (as a "bare" type) to IL's $(\mathbf{1} \vee \mathbf{1})$, while the closest LL equivalent, $\mathbf{1} \oplus \mathbf{1}$, maps to the IL type $(\neg\neg\mathbf{1} \vee \neg\neg\mathbf{1})$. Accommodating Haskell types requires a more complex translation, which, however, introduces no new difficulties.

The representation of functions is interesting. Logical implication $P \Rightarrow Q$ is classically equivalent to $\neg P \vee Q$, but this will not work as a function type in our intuitionistic calculus. An IL value of type $\neg P \vee Q$ is either a value from $\neg P$ or a value from Q; the former includes all divergent functions and the latter all constant functions, but there are no values which can accept an argument and return an answer that depends on that argument. The type $\neg(P \,\&\, \neg Q)$, again classically equivalent to implication, does not share this problem. Its operational interpretation is that a function is a continuation which takes two values, one of type P (the argument) and one of type $\neg Q$ (somewhere to send the result). This is exactly how function calls work in practical abstract machines: the two arguments to this continuation are the two values—argument and return address—pushed onto the stack before jumping to the function's entry point.

The translations from SL and LL terms to IL terms are shown in Fig. 7. These are the familiar continuation-passing translations of Plotkin [6]. Because

(In the LL to IL translation $\llbracket e \rrbracket_v$ abbreviates $\lambda k.\ \llbracket e \rrbracket \rhd k$)

SL/LL term	Translation (SL to IL)	Translation (LL to IL)
$\llbracket x \rrbracket \rhd E$	$E \bowtie x$	$x \bowtie E$
$\llbracket e\, e' \rrbracket \rhd E$	$\llbracket e \rrbracket \rhd \lambda x.\ \llbracket e' \rrbracket \rhd \lambda x'.\, x \bowtie (x', E)$	$\llbracket e \rrbracket \rhd \lambda x.\, x \bowtie (\llbracket e' \rrbracket_v, E)$
$\llbracket (e, e') \rrbracket \rhd E$	$\llbracket e \rrbracket \rhd \lambda x.\ \llbracket e' \rrbracket \rhd \lambda x'.\, E \bowtie (x, x')$	$E \bowtie (\llbracket e \rrbracket_v, \llbracket e' \rrbracket_v)$
$\llbracket \mathbf{inl}\ e \rrbracket \rhd E$	$\llbracket e \rrbracket \rhd \lambda x.\, E \bowtie \mathbf{inl}\ x$	$E \bowtie \mathbf{inl}\ \llbracket e \rrbracket_v$
$\llbracket \mathbf{inr}\ e \rrbracket \rhd E$	$\llbracket e \rrbracket \rhd \lambda x.\, E \bowtie \mathbf{inr}\ x$	$E \bowtie \mathbf{inr}\ \llbracket e \rrbracket_v$
$\llbracket \lambda x.\, e \rrbracket \rhd E$	$E \bowtie \lambda(x, k).\ \llbracket e \rrbracket \rhd k$	
$\llbracket () \rrbracket \rhd E$	$E \bowtie ()$	
$\llbracket \mathbf{error} \rrbracket \rhd E$	**diverge**	
$\llbracket \mathbf{case}\, e_1\ \{(x)e_2; (y)e_3\} \rrbracket \rhd E$	$\llbracket e_1 \rrbracket \rhd \lambda z.\, \mathbf{case}\ z\, \{(x)\, \llbracket e_2 \rrbracket \rhd E; (y)\, \llbracket e_3 \rrbracket \rhd E\}$	
$\llbracket \mathbf{case}\, e_1\ \{(x, y)e_2\} \rrbracket \rhd E$	$\llbracket e_1 \rrbracket \rhd \lambda(x, y).\ \llbracket e_2 \rrbracket \rhd E$	

Fig. 7. Translation of SL/LL to IL terms (type signatures omitted)

IL has explicit continuations while the continuations in SL/LL are implicit, the translation must be done in the context of an IL continuation. $\llbracket e \rrbracket \rhd E$ denotes the IL expression which passes e's value on to the IL continuation E. This should be treated as a syntactic unit; $\rhd$ has no meaning on its own. For LL we write $\llbracket e \rrbracket_v$ as a shorthand for $\lambda k.\ \llbracket e \rrbracket \rhd k$; this notation will prove useful in Section 4.2.

Type signatures have been omitted for space and readability reasons. Restoring the type signatures and adding standard typing rules for SL/LL terms, it can be shown that the translation of a well-typed SL/LL term is a well-typed IL term. In fact, we can show that if $e : t$ and $(\llbracket e \rrbracket \rhd E) : \mathbf{0}$, then $E : \mathcal{D}_k \llbracket t \rrbracket$, and (in LL) that if $e : t$, then $\llbracket e \rrbracket_v : \mathcal{D}_v \llbracket t \rrbracket$. Note the translations of $\llbracket x \rrbracket \rhd E$, which capture the essential difference between "ML-like" and "Haskell-like" embeddings in IL.

Translation Examples. For a first example we consider the SL/LL expression **inl error**. In SL this expression will clearly always diverge when evaluated, and our SL-to-IL translation turns out to yield **diverge** directly:

$$\llbracket \mathbf{inr\ error} \rrbracket \rhd k \quad = \quad \llbracket \mathbf{error} \rrbracket \rhd \lambda x.\, k \bowtie \mathbf{inr}\ x \quad = \quad \mathbf{diverge}$$

The LL-to-IL translation instead boxes the divergence:

$$\llbracket \mathbf{inr\ error} \rrbracket \rhd k \;=\; k \bowtie \mathbf{inr}\ (\lambda k'.\ \llbracket \mathbf{error} \rrbracket \rhd k') \;=\; k \bowtie \mathbf{inr}\ (\lambda k'.\, \mathbf{diverge})$$

The translations of the nested function application $p\,(q\,r)$ are interesting. From SL we have the following translation, where $\overset{*}{=}$ denotes a sequence of several (trivial) translation steps, and $\overset{*}{\rightsquigarrow}$ denotes a sequence of "clean-up" beta reductions after the translation proper.

$$\begin{aligned}
\llbracket p\,(q\,r) \rrbracket \rhd k \;&=\; \llbracket p \rrbracket \rhd \lambda f.\ \llbracket q\,r \rrbracket \rhd \lambda x.\, f \bowtie (x, k) \\
&=\; \llbracket p \rrbracket \rhd \lambda f.\ \llbracket q \rrbracket \rhd \lambda f'.\ \llbracket r \rrbracket \rhd \lambda x'.\, f' \bowtie (x', \lambda x.\, f \bowtie (x, k)) \\
&\overset{*}{=}\; (\lambda f.\, (\lambda f'.\, (\lambda x'.\, f' \bowtie (x', \lambda x.\, f \bowtie (x, k))) \bowtie r) \bowtie q) \bowtie p \\
&\overset{*}{\rightsquigarrow}\; q \bowtie (r, (\lambda x.\, p \bowtie (x, k)))
\end{aligned}$$

For LL we have

$$\begin{aligned}
\llbracket p\,(q\,r) \rrbracket \rhd k &= \llbracket p \rrbracket \rhd \lambda f.\, f \bowtie ((\lambda k'.\ \llbracket q\,r \rrbracket \rhd k'), k) \\
&= \llbracket p \rrbracket \rhd \lambda f.\, f \bowtie ((\lambda k'.\ \llbracket q \rrbracket \rhd \lambda f'.\, f' \bowtie ((\lambda k''.\ \llbracket r \rrbracket \rhd k''), k')), k) \\
&\overset{*}{=} p \bowtie \lambda f.\, f \bowtie ((\lambda k'.\, q \bowtie (\lambda f'.\, f' \bowtie ((\lambda k''.\, r \bowtie k''), k'))), k)
\end{aligned}$$

Our operational semantics cannot simplify this term. But it can be shown that η reduction is safe in IL (in the sense of contextual equivalence), so we may reduce it to $p \bowtie \lambda f.\, f \bowtie ((\lambda k'.\, q \bowtie (\lambda f'.\, f' \bowtie (r, k'))), k)$. Using the **lift** operation from section 2.3, and renaming k' to x, we get $p \bowtie \mathbf{lift}\,((\lambda x.\, q \bowtie \mathbf{lift}\,(r, x)), k)$, which, modulo lifting, is surprisingly similar to its SL counterpart.

4 AM

AM is an abstract machine designed to run IL code. The primitives of AM are chosen to resemble machine-code or byte-code instructions. AM is untyped for reasons of simplicity.

The purpose of AM is to provide a framework for discussing the low-level optimizations that make subtyping possible, which are described in Section 5. We are not concerned with a formal treatment of compilation as such, nor are we interested in most of the optimizations found in existing abstract machines. Therefore we will define AM only informally, and will gloss over most performance issues.

AM is a register machine. Throughout this section register names will be written in `typewriter face`. There are two special registers `env` and `arg`, which are used when entering a continuation, as well as a collection of compile-time constants, which are never assigned to, but are otherwise indistinguishable from registers. All other registers are local temporaries. There is never a need to save registers across function calls, because every call is a tail call. AM has no stack.

Registers hold *machine words*, which can be pointers to heap objects, pointers to addressable code, or tag values (which will be discussed later).

There are just five instructions in AM:

$M ::= \texttt{x} \leftarrow \texttt{y}$	Sets register `x` equal to register `y`.
$\mid \texttt{x} \leftarrow \texttt{y}[i]$	Indexed load: Register `x` gets the value at offset i within the heap object pointed to by register `y`.
$\mid \texttt{x} \leftarrow \textbf{new}\ \texttt{y1}, \ldots, \texttt{yn}$	Allocates n consecutive words from the heap, places the address of the allocated memory in register `x`, and stores the operands $\texttt{y1}, \ldots, \texttt{yn}$ at locations $\texttt{x}[0], \ldots, \texttt{x}[n-1]$.
$\mid \textbf{if}\ \texttt{x1} = \texttt{x2}$ $\textbf{then}\ M^*$ $\textbf{else}\ M^*$	Compares two registers and executes the first instruction sequence if they are equal, the second instruction sequence if they are not equal. M^* denotes a sequence of zero or more instructions.
$\mid \textbf{jump}\ \texttt{x}$	Transfers control to the instruction sequence whose address is in register `x`.

To make code more concise, we allow indexed-load expressions $\texttt{r}[i]$ wherever a register operand is expected. For example, the instruction $\textbf{jump}\ \texttt{env}[0]$ is equivalent to the two-instruction sequence $\texttt{tmp} \leftarrow \texttt{env}[0]\ ;\ \textbf{jump}\ \texttt{tmp}$.

While IL is indifferent as regards evaluation order, we choose to use eager evaluation when we compile it to AM.

4.1 Compilation

We have already noted that IL expressions divide naturally into those of type **0** and those not of type **0**. In AM this has the following concrete meaning:

- IL expressions of type **0** compile to *addressable instruction sequences*. These have an entry point which we jump to when calling a continuation, and they terminate by jumping to another addressable instruction sequence.
- IL expressions of other types compile to *non-addressable instruction sequences*: these appear within addressable instruction sequences and construct values on the heap.

Compilation form	Expansion
$\mathcal{F}\,[\![E_1 \bowtie E_2]\!]\ \mathit{\Gamma}$	$\mathcal{E}\,[\![E_1]\!]\ \mathit{\Gamma}\ \mathtt{tmp1}\ ;\ \mathcal{E}\,[\![E_2]\!]\ \mathit{\Gamma}\ \mathtt{tmp2}$ $\mathtt{env} \leftarrow \mathtt{tmp1}\ ;\ \mathtt{arg} \leftarrow \mathtt{tmp2}$ $\mathbf{jump}\ \mathtt{env}[0]$
$\mathcal{F}\,[\![\mathbf{case}\,E\,\{(x)F_1; (y)F_2\}]\!]\ \mathit{\Gamma}$	$\mathcal{E}\,[\![E]\!]\ \mathit{\Gamma}\ \mathtt{tmp}$ $\mathbf{if}\ \mathtt{tmp}[0] = \mathtt{tagLeft}$ $\quad\mathbf{then}\ \mathcal{F}\,[\![F_1]\!]\ (\mathit{\Gamma}[\mathtt{tmp}[1]/x])$ $\quad\mathbf{else}\ \ \mathcal{F}\,[\![F_2]\!]\ (\mathit{\Gamma}[\mathtt{tmp}[1]/y])$
$\mathcal{E}\,[\![x]\!]\ \mathit{\Gamma}\ \mathtt{r}$	$\mathtt{r} \leftarrow \mathit{\Gamma}(x)$
$\mathcal{E}\,[\![()]\!]\ \mathit{\Gamma}\ \mathtt{r}$	$\mathtt{r} \leftarrow \mathbf{new}\ \mathtt{tagUnit}$
$\mathcal{E}\,[\![(E_1, E_2)]\!]\ \mathit{\Gamma}\ \mathtt{r}$	$\mathcal{E}\,[\![E_1]\!]\ \mathit{\Gamma}\ \mathtt{tmp1}\ ;\ \mathcal{E}\,[\![E_2]\!]\ \mathit{\Gamma}\ \mathtt{tmp2}$ $\mathtt{r} \leftarrow \mathbf{new}\ \mathtt{tagPair}, \mathtt{tmp1}, \mathtt{tmp2}$
$\mathcal{E}\,[\![\mathbf{inl}\ E]\!]\ \mathit{\Gamma}\ \mathtt{r}$	$\mathcal{E}\,[\![E]\!]\ \mathit{\Gamma}\ \mathtt{tmp}\ ;\ \mathtt{r} \leftarrow \mathbf{new}\ \mathtt{tagLeft}, \mathtt{tmp}$
$\mathcal{E}\,[\![\mathbf{inr}\ E]\!]\ \mathit{\Gamma}\ \mathtt{r}$	$\mathcal{E}\,[\![E]\!]\ \mathit{\Gamma}\ \mathtt{tmp}\ ;\ \mathtt{r} \leftarrow \mathbf{new}\ \mathtt{tagRight}, \mathtt{tmp}$
$\mathcal{E}\,[\![\mathbf{fst}\ E]\!]\ \mathit{\Gamma}\ \mathtt{r}$	$\mathcal{E}\,[\![E]\!]\ \mathit{\Gamma}\ \mathtt{tmp}\ ;\ \mathtt{r} \leftarrow \mathtt{tmp}[1]$
$\mathcal{E}\,[\![\mathbf{snd}\ E]\!]\ \mathit{\Gamma}\ \mathtt{r}$	$\mathcal{E}\,[\![E]\!]\ \mathit{\Gamma}\ \mathtt{tmp}\ ;\ \mathtt{r} \leftarrow \mathtt{tmp}[2]$
$\mathcal{E}\,[\![\lambda(x\!:\!T).\,F]\!]\ \mathit{\Gamma}\ \mathtt{r}$	$\mathtt{r} \leftarrow \mathbf{new}\ \mathtt{code}, \mathit{\Gamma}(v_1), \ldots, \mathit{\Gamma}(v_n)$ *(see text)*
$\mathcal{E}\,[\![\mathbf{case}\,E_1\,\{(x)E_2; (y)E_3\}]\!]\ \mathit{\Gamma}\ \mathtt{r}$	$\mathcal{E}\,[\![E_1]\!]\ \mathit{\Gamma}\ \mathtt{tmp}$ $\mathbf{if}\ \mathtt{tmp}[0] = \mathtt{tagLeft}$ $\quad\mathbf{then}\ \mathcal{E}\,[\![E_2]\!]\ (\mathit{\Gamma}[\mathtt{tmp}[1]/x])\ \mathtt{r}$ $\quad\mathbf{else}\ \ \mathcal{E}\,[\![E_3]\!]\ (\mathit{\Gamma}[\mathtt{tmp}[1]/y])\ \mathtt{r}$

Fig. 8. AM compilation rules

Fig. 8 lists the rules for compiling an IL expression to an instruction sequence. There is a separate set of rules for non-**0** types ($\mathcal{E}$) and type **0** ($\mathcal{F}$). The $\mathcal{E}$ rules take an extra parameter `r`, the register which receives the result value.

Within the context of each expansion, register names beginning `tmp` are instantiated with fresh temporary register names. Each such register is assigned to exactly once; thus the code is in SSA form as regards temporary registers. This does not apply to the special registers `env` and `arg`, which are overwritten just before each **jump** instruction. Note that there is no need to save the old values of `env` and `arg` or of any temporary register, since continuations never return.

When expanding $\mathcal{E}\,[\![\lambda x.\,F]\!]\ \mathit{\Gamma}\ \mathtt{r}$, the compiler also expands

$$\mathcal{F}\,[\![F]\!]\ (x \mapsto \mathtt{arg}, y_1 \mapsto \mathtt{env}[1], \ldots, y_n \mapsto \mathtt{env}[n]),$$

where $\{y_1, \ldots, y_n\} = \mathrm{fv}(\lambda x.\,F)$, and places the expanded code somewhere in the code segment. The local (fresh) constant register `code` is set to the entry point of this code.

4.2 Updating

There is one optimization that every non-strict language implementation must perform, because compiled code may run exponentially slower without it. This is *thunk memoization* or *thunk updating*, and it is the difference between lazy evaluation and normal-order reduction. It is described, for example, in [4]. The details of this process are ugly, and a full treatment would complicate IL and AM significantly; but we cannot ignore it entirely, since it interacts non-trivially with the subtyping system of the next section.

For our purposes, a *thunk* is a heap object constructed by executing $\mathcal{E}\,\llbracket E \rrbracket\,\Gamma\,r$, where E was produced by the $\llbracket e \rrbracket_v$ rule during translation from LL to IL. Such an expression has the form $ƛ(k : \neg T).\,F$, where F either diverges or computes a value V and passes it to k. If F successfully computes a value, then before passing that value to k we must *update* the thunk by physically overwriting its heap representation with a new object equivalent to $(ƛ(k{:}\neg T).\,k \bowtie V)$. The new object is indistinguishable from the old as far as the programmer is concerned: we know, by virtue of having just evaluated F, that its effect is just $k \bowtie V$. (And since IL is referentially transparent it cannot have any side effect.)

We might model updating by extending AM with a new instruction

M ::= . . .
  | $\mathtt{x}[i] \leftarrow \mathtt{y}$ Indexed store: Overwrites the word at index i in the heap object pointed to by `x` with the word in `y`.

in terms of which the updating step may be written $\mathtt{thunk}[0] \leftarrow \mathtt{ret\_payload}$; $\mathtt{thunk}[1] \leftarrow \mathtt{val}$, where `thunk` points to the heap object to be updated, `val` points to the heap representation of V, and `ret_payload` points to the code $\mathcal{F}\,\llbracket k \bowtie v \rrbracket\;(k \mapsto \mathtt{arg}, v \mapsto \mathtt{env}[1])$. When `ret_payload` is called, $\mathtt{env}[1]$ will contain the computed value that we stored in $\mathtt{thunk}[1]$. This form of updating is similar to updating with an *indirection node* in GHC [4].

5 Subtyping and Auto-Lifting

It turns out that with a small variation in the design of AM, the natural embedding from unlifted to lifted types becomes a subtyping relationship, allowing us to treat any ML value as a Haskell value at runtime without cost, and project a Haskell value onto an ML type with only the cost of a Haskell `deepSeq`.

Suppose that we have a value V of type T, and wish to construct a value V' of type $\neg\neg T$, most likely for the purpose of marshalling data from eager to lazy code. If $T = (T_1 \,\&\, T_2)$, then V can only be (V_1, V_2) for some values V_1 and V_2. Then $V' = ƛk.\,k \bowtie (V_1, V_2)$. But we cannot compile a fresh addressable instruction sequence $k \bowtie (V_1, V_2)$ for each V_1 and V_2, since V_1 and V_2 are not known at compile time. Instead we compile a single instruction sequence $\mathcal{F}\,\llbracket k \bowtie (v_1, v_2) \rrbracket\;(k \mapsto \mathtt{arg}, v_1 \mapsto \mathtt{env}[1], v_2 \mapsto \mathtt{env}[2])$ and place pointers to V_1 and V_2 in the appropriate environment slots at run time.

Similarly, if $T = (T_1 \vee T_2)$, then V is $\mathbf{inl}\,V_1$ or $\mathbf{inr}\,V_2$, so we compile $\mathcal{F}\,\llbracket k \bowtie \mathbf{inl}\;v_1 \rrbracket\;(k \mapsto \mathtt{arg}, v_1 \mapsto \mathtt{env}[1])$ and $\mathcal{F}\,\llbracket k \bowtie \mathbf{inr}\;v_2 \rrbracket\;(k \mapsto \mathtt{arg}, v_2 \mapsto \mathtt{env}[1])$, and place V_1 or V_2 in the environment slot.

In short, the lifted pair $(ƛk.\,k \bowtie (V_1, V_2))$ will be represented on the heap by an object of three words, the first being a pointer to the code for $k \bowtie (v_1, v_2)$ and the second and third being pointers to V_1 and V_2. The lifted left injection will be represented by an object of two words, the first being a pointer to the code for $k \bowtie \textbf{inl}\ v_1$ and the second being a pointer to V_1; and similarly for the right injection. These heap objects are the same size as the heap objects we would construct for the unlifted values V; except for the first word, the layout is the same; and since we compiled just three instruction sequences, the first word of the lifted values can contain just three different pointers, which are in one-to-one correspondence with the three tags `tagPair`, `tagLeft`, `tagRight`. So if we simply *define* our sum and product tags to be pointers to the appropriate instruction sequence, then a heap representation of any value of an IL sum or product type is also a value of that type's double negation. We will call this *auto-lifting.*

Auto-lifting also works for `tagUnit`, but a slightly different approach is needed for function types, and more generally for any type beginning with $\neg$. Further discussion of this may be found at the web site [7].

An obvious but nonetheless interesting observation about auto-lifting is that it is often *polynomially* faster than explicit lifting. Explicitly converting from a strict list to a non-strict list in IL is $\Theta(n)$ in the size of the list, while auto-lifting is free of cost independently of n.

5.1 Coercing LL Values to SL Values

The function `deepSeq` is not built in to Haskell, but can be defined using type classes. Its operational effect is to traverse a data structure, *forcing* each node as it goes. By forcing we mean, in IL terms, calling each continuation $\neg\neg T$ and ignoring the result T (except for purposes of further traversal). Applying `deepSeq` to a data structure has the referentially transparent "side effect" of causing all nodes in the data structure to be updated with values of the form $\lambda k.\, k \bowtie V$ (see Section 4.2). If there is no such value—if $\perp$ is hiding anywhere in the data structure—then `deepSeq` diverges.

We have already arranged that a fully-evaluated LL value *is* an SL value in AM. It would seem that if we define our forcing operation in such a way that it overwrites thunks with valid SL values, then provided `deepSeq` does not diverge, we could subsequently treat its argument as having the corresponding SL type.

Unfortunately, this does not quite work. The trouble is that we cannot always overwrite a thunk with a valid SL value. Consider, for example, the thunk $\lambda k.\, k \bowtie (x, x)$. This has one free variable (x), and so its heap representation in AM occupies two words (the other being the code pointer). Its SL counterpart, on the other hand, requires *three* words (one for the tag and two for the fields of the pair). We can solve this in some cases by setting aside extra space when we allocate the thunk, but this is not always possible in a practical implementation with larger tuples and polymorphism. To handle the remaining cases, we are forced to (re-)introduce indirection nodes. But indirection nodes are not SL values!

Fortunately, the solution is not difficult. We must think of `deepSeq` not as a procedure but as a function that returns an SL value as its result. If in-place updating is possible, `deepSeq` returns its argument (which is then a valid SL value); if in-place updating is not possible, `deepSeq` updates with an indirection and returns the *target* of that indirection, which is again a valid SL value.

A related complication arises when we use deepSeq on a data structure with a mixture of strict and non-strict fields, such as might be defined in a hybrid language. In such cases we must update not only thunks but also the fields of SL values. Because of space constraints we do not discuss the details.

6 Conclusions

In this paper we defined an intermediate language IL, containing continuations but not functions, which can encode naturally both strict and non-strict languages; and we exhibited an abstract machine, AM, which can execute IL (and, via translation, strict and non-strict source languages) with an efficiency comparable to existing ML and Haskell implementations modulo known optimization techniques, while also supporting efficient interconversion between ML data structures and Haskell data structures.

IL seems to capture fundamental aspects of the relationship between strict and non-strict languages which we had previously understood only in an ad hoc manner. The fact that a structure resembling a lifting monad appears within IL, without any attempt to place it there (Section 2.3) is one example of this. In fact IL's three negation classes do the lifting monad one better, since they predict that lifting leads to a semantic distinction (in the sense of Section 2.3) only in a value context, not in an expression (continuation) context. It follows that, in a hybrid strict/lazy language, it makes sense to annotate the strictness of function *arguments*, but not of function *results*—a fact that we recognized long before discovering IL, but for which we had never had a satisfactory theoretical model. In this and other ways IL seems to be predictive where previous systems were phenomenological, and this is its primary appeal.

6.1 Related Work

The benefits of continuation-passing style for compilation, and the existence of call-by-name and call-by-value CPS translations, have been known for decades [6]. The notion of continuations as negations was introduced by Griffin [3]. Recently several authors have introduced computational calculi to demonstrate the call-by-value/call-by-name duality within a classical framework, including Curien and Herbelin's lambda-bar calculus [2], Wadler's dual calculus [9], and van Bakel, Lengrand, and Lescanne's $\mathcal{X}$ [8]. Wadler explicitly defines the function arrow in terms of negation, conjunction and disjunction. On the practical side, a previous paper by Peyton Jones, Launchbury, Shields, and Tolmach [5] studies the same practical problem as the present work, proposing a monad-based intermediate language also called IL.

The present work represents, we believe, a happy medium between the theoretical and practical sides of the problem we set out to solve. IL is straightforwardly and efficiently implementable on stock hardware, while retaining some of the interesting features of its theoretical cousins; it is also substantially simpler than the previous proposal by Peyton Jones et al, while preserving its fundamental design goal. The notion of auto-lifting described in this paper may also be new to the literature, though it was known to the authors before this research began.

6.2 Future Work

The work described in this paper is part of an ongoing research project. Again we invite interested readers to visit the project web site [7], which will contain additional material omitted from this paper as well as updates on further progress.

As of this writing, and undoubtedly as of publication time, a large amount of work remains to be done. We have not implemented a compiler based on IL and AM, and many issues must be investigated and resolved before we can do so. Some optimizations can be accommodated quite well within IL as it stands—for example, the "vectored return" optimization of the STG-machine [4] is valid as a consequence of de Morgan's law. Others require further work. The minimalist design of AM can accommodate extensions for stack-based evaluation, register arguments and the like, if these can be represented neatly in IL. Since the use of continuation-passing calculi as intermediate languages is well understood [1], it seems likely that this can be done using known techniques.

Adding polymorphism to IL is not difficult, and the translation from the source language to IL remains straightforward as long as the source language is strict or non-strict. Unfortunately, attempts to introduce unrestricted polymorphism into a hybrid language lead to subtle difficulties. IL does not cause these difficulties, but only exposes them; we hope that further study of IL will expose a sensible solution as well.

Acknowledgements

This work was supported by a studentship from Microsoft Research Cambridge. We are grateful to Philip Wadler and the anonymous reviewers for helpful comments.

References

1. Andrew W. Appel, *Compiling With Continuations.* Cambridge University Press, 1992. ISBN 0–521–41695–7.
2. Pierre-Louis Curien and Hugo Herbelin, *The duality of computation.* Proc. ICFP 2000. `http://pauillac.inria.fr/~herbelin/habilitation/icfp-CurHer00-duality+errata.ps`
3. Timothy G Griffin, *A formulae-as-types notion of control.* Proc. POPL 1990.
4. Simon Peyton Jones, *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine.* Journal of Functional Programming, 2(2):127–202, 1992.

5. Simon Peyton Jones, John Launchbury, Mark Shields, and Andrew Tolmach, *Bridging the gulf: a common intermediate language for ML and Haskell*. Proc. POPL 1998. `http://www.cartesianclosed.com/pub/intermediate_language/neutral.ps`
6. Gordon D. Plotkin, *Call-by-name, call-by-value and the λ-calculus*. Theoretical Computer Science 1:125–159, 1975. `http://homepages.inf.ed.ac.uk/gdp/publications/cbn_cbv_lambda.pdf`
7. Ben Rudiak-Gould, *The NotNotML project website*. `http://www.cl.cam.ac.uk/~br276/notnotML/`
8. Steffen van Bakel, Stéphane Lengrand, and Pierre Lescanne, *The language $\mathcal{X}$: circuits, computations and Classical Logic*. Proc. ICTCS 2005. `http://www.doc.ic.ac.uk/~svb/Research/Abstracts/vBLL.html`
9. Philip Wadler, *Call-by-value is dual to call-by-name*. Proc. ICFP 2003, pp. 189–201. `http://homepages.inf.ed.ac.uk/wadler/papers/dual/dual.pdf`