Scrap your type applications

Barry Jay[1] and Simon Peyton Jones[2]

[1] University of Technology, Sydney
2 Microsoft Research Cambridge

Abstract. System **F** is ubiquitous in logic, theorem proving, language meta-theory, compiler intermediate languages, and elsewhere. Along with its type abstractions come *type applications*, but these often appear redundant. This redundancy is both distracting and costly for type-directed compilers.
We introduce System **IF**, for *implicit* System **F**, in which many type applications can be made implicit. It supports decidable type checking and strong normalisation. Experiments with Haskell suggest that it could be used to reduce the amount of intermediate code in compilers that employ System **F**.
System **IF** constitutes a first foray into a new area in the design space of typed lambda calculi, that is interesting in its own right and may prove useful in practice.
This paper appears in the Proceedings of Mathematics of Program Construction (MPC'08), Marseille, July 2008.

1 Introduction

The polymorphic lambda calculus or System **F** is ubiquitous in many areas of computer science such as logic, e.g. (Girard et al. 1989; Girard 1990), programming, e.g. (Reynolds 1974), theorem-proving, e.g. (Coq), and intermediate languages for compilers, e.g. (Peyton Jones 2003; Harper and Morrisett 1995). System **F** is, however, tiresomely verbose. For example, suppose the first projection from a pair is given by `fst` : $\forall a.\forall b.(a, b) \to a$. Then the first projection of the pair `(3,True)` is given by

```
fst Int Bool (3,True)
```

where the two type arguments, `Int` and `Bool`, are required to instantiate the type variables a and b. To a naive observer, the type arguments seem redundant. After all, if we were to write simply

```
fst (3,True)
```

then it is clear how to instantiate a and b! And indeed many *source* languages omit type arguments, relying on type inference to fill them in. However our interest is in typed *calculi* with the power of System F, for which type inference known to be undecidable (Wells 1994). More precisely, we address the following question: can we omit type arguments in a polymorphic calculus with the

full expressive power of System **F**, without losing decidable type checking? Our contributions are as follows:

- We present a new, explicitly-typed lambda calculus, System **IF** (short for "implicit System **F**") that is precisely as expressive as System **F**, but allows many type application to be be scrapped (Section 3). However, it requires four new reduction rules.
- System **IF** enjoys the same desirable properties as System **F**; in particular, type checking is decidable, and reduction is type preserving, confluent, and strongly normalising (Section 3.4). Furthermore, **IF** shares System **F**'s property that every term has a unique type (Section 2.2), which is particularly useful when the calculus is used as intermediate language for a compiler (Section 4.1).
- Every System **F** term is also an System **IF** term; and conversely there is translation from System **IF** to System **F** that preserves typing, type erasure, term equality and inequality (Section 3.3). Reduction itself is not preserved since one reduction rule is reversed during translation.
- We regard System **IF** as of interest in its own right, but it potentially has some practical importance because compilers for higher-order, typed languages often use an explicitly-typed intermediate language based on System **F** (Peyton Jones et al. 1993; Tarditi et al. 1996; Shao 1997), and there is evidence that cost of processing types is a significant problem in practice (Shao et al. 1998; Petersen 2005). To get some idea of whether System **IF** is useful in this context, we adapted the Glasgow Haskell Compiler (GHC), a state-of-the-art compiler for Haskell, to use System **IF** as its intermediate language (Section 4). The results are mixed: 80% of all type applications can be removed, reducing the total size of the code by 12%, but the "bottom line" of compiler execution time is not improved (Section 4).

Our work complements other approaches that reduce the burden of type information in intermediate languages (Section 5), but it is distinctively different: to the best of our knowledge no previous such work specifically considers how to eliminate type applications.

Although the emphasis in this paper is on intermediate languages, the ideas may be of broader significance. In programming, for example, quantified function types allow for the possibility of combining cases whose types quantify different numbers of type variables. In each case the instantiation of type variables will be determined by the argument, not by the programmer, in a form of dynamic dispatch for type variables (Jay 2006).

2 System F

This section recalls System **F** and discusses its redundant type applications.

Syntax

$$
\begin{array}{rl}
a, b, c & ::= \langle \text{type variables} \rangle \\
f, g, x, y, z & ::= \langle \text{term variables} \rangle \\
\\
\sigma, \tau, \phi, \psi & ::= a \mid \sigma \to \sigma \mid \forall a.\sigma \\
r, s, t, u & ::= x^{\sigma} \mid t\ t \mid t\ \sigma \mid \lambda x^{\sigma}.t \mid \Lambda a.t \\
\Gamma & ::= x_1^{\sigma_1}, \ldots, x_n^{\sigma_n} \quad (n \geq 0) \\
\Delta & ::= a_1, \ldots, a_n \quad (n \geq 0,\ a_i \text{ distinct})
\end{array}
$$

Notation

$$
\begin{array}{rl}
\forall \Delta.\sigma & \equiv \forall a_1 \ldots \forall a_n.\sigma \\
\Lambda \Delta.t & \equiv \Lambda a_1 \ldots \Lambda a_n.t \\
t\ \Delta & \equiv t\ a_1 \ldots a_n \\
t\ (\theta\ \Delta) & \equiv t\ (\theta a_1)\ \ldots\ (\theta a_n) \quad \text{for } \theta \text{ a type substitution} \\
\mathsf{FTV}(\sigma) & \equiv \text{the free type variables of } \sigma
\end{array}
$$

Type system

$$\text{(fvar)}\ \frac{}{\Gamma, x^{\sigma} \vdash x^{\sigma} : \sigma}$$

$$\text{(fapp)}\ \frac{\Gamma \vdash r : \sigma \to \phi \qquad \Gamma \vdash u : \sigma}{\Gamma \vdash r\ u : \phi} \qquad \text{(fabs)}\ \frac{\Gamma, x^{\sigma} \vdash s : \phi \quad x \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash \lambda x^{\sigma}.s : \sigma \to \phi}$$

$$\text{(tapp)}\ \frac{\Gamma \vdash t : \forall a.\sigma}{\Gamma \vdash t\ \psi : \{\psi/a\}\sigma} \qquad \text{(tabs)}\ \frac{\Gamma \vdash s : \sigma \quad a \notin \mathsf{FTV}(\Gamma)}{\Gamma \vdash \Lambda a.s : \forall a.\sigma}$$

Dynamic semantics

$$
\begin{array}{rl}
(\beta 1) & (\lambda x^{\sigma}.s)\ u \longrightarrow \{u/x\}s \\
(\beta 2) & (\Lambda a.s)\ \psi \longrightarrow \{\psi/a\}s
\end{array}
$$

Fig. 1: Definition of System **F**

2.1 The system

The syntax, notation, type system and dynamic semantics for System **F** are given for easy reference in Figure 1. They should be familiar, but they serve to establish our notational conventions.

The standard definitions apply for: the free type variables $\mathsf{FTV}(\sigma)$ of a type σ; the free type variables $\mathsf{FTV}(t)$ of a term t; the free term variables $\mathsf{ftv}(t)$ of t; type and term substitutions; and the α-equivalence relations for re-naming bound type and term variables. The notation $\{\sigma_1/a_1, \ldots, \sigma_n/a_n\}$ is the substitution mapping a_i to σ_i, and $\{\sigma_1/a_1, \ldots, \sigma_n/a_n\}\phi$ is the result of applying that substitution to the type ϕ. (And similarly for terms.) A type environment Γ is a partial function of finite domain, from term variables to types. The notation $x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}$ may be used to denote the function that maps x_i to σ_i. The domain of Γ is denoted $\mathsf{dom}(\Gamma)$.

The symbol Δ stands for a sequence of distinct type variables $a_1, \ldots, a_n$, and we may write $\forall\Delta.\sigma$ to abbreviate $\forall a_1 \ldots \forall a_n.\sigma$. Similar syntactic sugar abbreviates type abstractions $\Lambda\Delta.t$ and type applications $t\ \Delta$ and even $t\ (\theta\Delta)$ for a type substitution θ.

The dynamic semantics is expressed using the reduction rules ($\beta 1$) and ($\beta 2$) which generate a rewriting relation in the usual way.

In our examples, we often use type constants `Int` and `Bool`, along with corresponding term constants `0`, `1`, ..., `True`, `False`, `(+)`, `(&&)`, and so on. We also assume a pair type (σ, ϕ), and a list type `List` σ; with term constructors

$$\begin{aligned}\texttt{Pair} &: \forall a, b.a \to b \to (a, b)\\ \texttt{Nil} &: \forall a.\texttt{List}\ a\\ \texttt{Cons} &: \forall a.a \to \texttt{List}\ a \to \texttt{List}\ a.\end{aligned}$$

Formally, these are just notations for their Church encodings, but no harm comes from considering them as extensions of the language.

2.2 Uniqueness of types

In Girard's original notation, each term variable is annotated with its type, at its occurrences as well as its binding site, but there are no type environments. Later treatments commonly include a type environment which enforces the invariant that every free occurrence of a term variable has the same type, an approach we follow in Figure 1. The type environment allows the type annotation to be dropped from term-variable occurrences, but we nevertheless retain them as a source of type information which will prove useful in the new system. For example, we write

$$(\lambda x^{\texttt{Int}}.\,\texttt{negate}^{\texttt{Int}\to\texttt{Int}}\ x^{\texttt{Int}}).$$

This notation looks somewhat verbose, in conflict with the goals of the paper, and indeed in our informal examples we often omit type attribution on variable occurrences. However, in practice there is no overhead to this apparent redundancy, as we discuss in Section 4.1, and it has an important practical benefit: every term has a unique type.

Theorem 1 (Uniqueness of types). *If $\Gamma \vdash t : \sigma$ and $\Gamma' \vdash t : \sigma'$ then $\sigma = \sigma'$.*

We can therefore write, without ambiguity, $t : \sigma$ or t^σ to mean that there is some Γ such that $\Gamma \vdash t : \sigma$.

This unique-type property is extremely convenient for a compiler that uses System **F** as an intermediate language. Why? Because every term has a unique type *independent of the context in which the term appears.* More concretely, the compiler may straightforwardly compute the type of any given term with a single, bottom-up traversal of the term, applying the appropriate rule of Figure 1 at each node of the term. System **IF** is carefully designed to retain this property.

2.3 Redundant type applications

Although System **F**'s *definition* is beautifully concise, its *terms* are rather verbose. This subsection will demonstrate this through some examples, as a motivation for the new systems that will follow.

Our particular focus is on *type applications*, the term form $(t\ \sigma)$. Human beings dislike writing type applications in source programs, because it is usually obvious what they should be, and they are burdensome to write. Rather, in programming languages such as ML and Haskell a type inference system, such as Hindley-Milner, fills them in.

Why are the missing type applications "obvious"? Because the types of the arguments of an application typically determine the appropriate type arguments. The introduction gave one example, but here is another. Suppose we are given terms $\texttt{map} : \forall a, b.(a \to b) \to \texttt{List}\ a \to \texttt{List}\ b$ and $\texttt{treble} : \texttt{Int} \to \texttt{Int}$. Then an application of `map` might look like this:

```
map Int Int treble
```

It is obvious that the type of `treble` immediately fixes both type arguments of `map` to be `Int`, so no information is lost by writing simply `(map treble)`.

The type arguments in System **F** can occasionally become onerous even to a computer. We encountered this in practice when implementing derivable type classes (Hinze and Peyton Jones 2000) in the Glasgow Haskell Compiler (GHC). The implementation transforms an N-ary data constructor into a nested tuple. For example, the Haskell term $(\texttt{C}\ e_1\ e_2\ e_3\ e_4\ e_5)$ is transformed to a nested tuple, whose System **F** representation looks like this (where $e_i : \sigma_i$):

$$\begin{array}{l}\texttt{Pair}\ \sigma_1\ (\sigma_2, (\sigma_3, (\sigma_4, \sigma_5)))\ e_1 \\ \quad (\texttt{Pair}\ \sigma_2\ (\sigma_3, (\sigma_4, \sigma_5))\ e_2 \\ \quad\quad (\texttt{Pair}\ \sigma_3\ (\sigma_4, \sigma_5)\ e_3\ (\texttt{Pair}\ \sigma_4\ \sigma_5\ e_4\ e_5))).\end{array}$$

Note the quadratic blow-up in the size of the term, because the type argument at each level of the nest repeats all the types already mentioned in its arguments. We are not the first to notice this problem, and we discuss in Section 5.2 ways of exploiting sharing to reduce the size of the term or of its representation. But it is more direct to simply *omit* the repeated types than to compress them! Furthermore, omitting them is entirely possible in this example: no information is lost by writing just

$$\texttt{Pair}\ e_1\ (\texttt{Pair}\ e_2\ (\texttt{Pair}\ e_3\ (\texttt{Pair}\ e_4\ e_5))).$$

Omitting obviously-redundant type applications in System **F** can therefore lead to an asymptotic reduction in the size of the term. When the calculus is used as an intermediate language in a compiler, reducing the size of terms may lead to improvements in compilation time.

This abbreviated form is, of course, exactly what an ML programmer would write, relying on type inference to fill in missing type arguments. So the reader

might wonder: why not simply omit type applications and use type inference to reconstruct them when necessary? We discuss this question in Section 5, but the short answer is this: type inference is undecidable for System **F** (Wells 1994). The trouble is that System **F** is far too expressive for type inference to work. In particular, in System **F** *type arguments may be quantified types*. For example, one might write:

$$\texttt{Pair}\ (\forall a.a \to a)\ \texttt{Int}\ (\Lambda a.\lambda x^a.\, x)\ 4$$

That is, System **F** is *impredicative*. Very few source languages are impredicative, but System **F** certainly is. Furthermore, a compiler that uses System **F** as its typed intermediate language may well exploit that expressiveness. For example, when desugaring mutually recursive bindings, GHC builds tuples whose components are polymporhic values, which in turn requires the tuple to be instantiated at those polytypes.

3 System IF

Syntax as for System **F**

Notation as for System **F** plus

$$\begin{aligned} \{\psi/[\Delta]\,\sigma\} &\equiv \text{the most general substitution } \theta \text{ (if any)} \\ &\qquad \text{such that } \mathsf{dom}(\theta) \subseteq \Delta \text{ and } \theta\sigma = \psi \\ \Delta\backslash\sigma &\equiv \Delta\backslash\mathsf{FTV}(\sigma) \end{aligned}$$

Type system as for System **F**, but replacing (fapp) by

$$\text{(ifapp)}\ \frac{\Gamma \vdash r : \forall\Delta.\sigma \to \phi \quad \Gamma \vdash u : \psi}{\Gamma \vdash r\ u : \forall(\Delta\backslash\sigma).\{\psi/[\Delta]\,\sigma\}\phi}\ \mathsf{FTV}(\psi) \cap (\Delta\backslash\sigma) = \emptyset$$

Dynamic semantics as for System **F**, plus

$$\begin{array}{lll} (\xi 1) & r^{\forall a,\Delta.\sigma\to\phi}\ \psi\ u \longrightarrow r\ u & \text{if } a \in \mathsf{FTV}(\sigma)\backslash\Delta \\ (\xi 2) & r^{\forall a,\Delta.\sigma\to\phi}\ \psi\ u \longrightarrow r\ u\ \psi & \text{if } a \notin \mathsf{FTV}(\sigma)\backslash\Delta \\ (\mu 1) & (\Lambda a.t^{\forall\Delta.\sigma\to\phi})\ u^{\psi} \longrightarrow \{\psi/[a,\Delta]\sigma\}t\ u & \text{if } a \in \mathsf{FTV}(\sigma)\backslash\Delta \\ & & \text{and } \mathsf{FTV}(t) \cap \Delta = \emptyset \\ (\mu 2) & (\Lambda a.t^{\forall\Delta.\sigma\to\phi})\ u^{\psi} \longrightarrow \Lambda a.(t\ u) & \text{if } a \notin \mathsf{FTV}(u) \cup (\mathsf{FTV}(\sigma)\backslash\Delta) \end{array}$$

Fig. 2: Definition of System **IF**

Thus motivated, we introduce System **IF**, short for "implicit System F", whose definition is given for reference in Figure 2. The key idea is this:

> System **IF** is just like System **F**, except that many type applications may be omitted.

In fact, System **F** is embedded in System **IF**: they have exactly the same types, the same term syntax, and every well-typed term in System **F** is a well-typed term in System **IF**.

But System **IF** has additional well-typed terms that System **F** lacks. In particular, a term of the form $r\ \psi_1\ \ldots\ \psi_n\ u$ in System **F** may be replaced by an equivalent term $r\ u$ of System **IF** provided that the types $\psi_1, \ldots, \psi_n$ can be recovered from the types of r and u. Of course, scrapping such type applications has knock-on effects on the type system and dynamic semantics.

3.1 Type system of IF

In System **F**, a term r of type $\forall a.\sigma$ can only be applied to a *type*, but in System **IF** it may also be applied to a *term*. The single new typing rule, (ifapp), specifies how such term applications are typed in System **IF**. The idea is this: if $r : \forall \Delta.\sigma \to \phi$ and $u : \psi$, then use ψ to instantiate all those type variables $a_i \in \Delta$ that appear free in σ.

For example, suppose $r : \forall a, b.(a \to \texttt{Int} \to b) \to \phi$ and $u : \texttt{Int} \to \texttt{Int} \to \texttt{Bool}$. Then if we see the application $(r\ u)$ it is plain that we must instantiate a to `Int` and b to `Bool`; and that no other instantiation will do. To derive this instantiation:

> we *match* $(a \to \texttt{Int} \to b)$ against $(\texttt{Int} \to \texttt{Int} \to \texttt{Bool})$
> to find a *substitution* for $\{a, b\}$
> namely $\{\texttt{Int}/a, \texttt{Bool}/b\}$.

(The notation $\{\psi/a, \tau/b\}$ stands for a substitution, as explained in Section 2.1.) More formally, we define a *match* of σ against ψ *relative to* a sequence of type variables Δ, written $\{\psi/[\Delta]\sigma\}$, to be a substitution θ whose domain is within Δ such that $\theta\sigma = \phi$. Such a match is *most general* if any other such match factors through it.

The rules for matching are straightforward, rather like those for unification, the only novelty being the treatment of quantified types. For example, to match $\forall d.(\forall b.b \to b) \to d$ against $\forall c.a \to c$, relative to a, first use α-conversion to identify c with d and then bind a to $\forall b.b \to b$. Note, however, that matching cannot employ such bound variables in either the range or domain of the substitution. For example, $\{\forall c.c \to a\,/[a]\,\forall c.c \to c\}$ fails, since one cannot substitute c for a under the quantifier $\forall c$; and similarly $\{\forall c.c \to c\,/[c]\,\forall c.Int \to c\}$ fails, since we cannot substitute for c under the quantifier.

These observations motivate the following definition: a type substitution υ *avoids* a type variable a if a is not in the domain or the range of υ.

Theorem 2. *If there is a match of type σ against a type ψ relative to a sequence of type variables Δ, then there is a most general such match. Further, there is at most one most general match, which is denoted $\{\psi/[\Delta]\sigma\}$.*

Proof. Now the most general match is defined as follows:

$$\begin{array}{rll}
\{\psi/[\Delta]a\} = \{\psi/a\} & & \text{if } a \in \Delta \\
\{a/[\Delta]a\} = \{\} & & \text{if } a \notin \Delta \\
\{\psi_1 \to \psi_2/[\Delta]\sigma_1 \to \sigma_2\} = & \text{let } \upsilon_1 = \{\psi_1/[\Delta]\sigma_1\} \text{ in} & \\
& \text{let } \upsilon_2 = \{\upsilon_1\psi_1/[\Delta]\upsilon_1\sigma_1\} \text{ in} & \\
& \upsilon_2 \circ \upsilon_1 & \\
\{\forall a.\psi/[\Delta]\forall a.\sigma\} = \{\psi/[\Delta]\sigma\} & & \text{if this avoids } a \\
\{\psi/[\Delta]\sigma\} = \text{undefined} & & \text{otherwise.}
\end{array}$$

The proof details are by straightforward induction. □

Now let us return to the typing of an application $(r\ u)$. What if the argument type of r does not mention all r's quantified type variables? For example, suppose $r : \forall a, b.(\texttt{Int} \to a) \to \phi$ and $u : \texttt{Int} \to \texttt{Bool}$. Then in the application $(r\ u)$ it is clear that we must instantiate a to `Bool`, but we learn nothing about the instantiation of b. In terms of matching, the match $\{\texttt{Int} \to \texttt{Bool}/[a, b]\texttt{Int} \to a\} = \{\texttt{Bool}/a\}$ does not act on b. That is, $(r\ u)$ is still polymorphic in b, which fact can be expressed by giving $(r\ u)$ the type $\forall b.\{\texttt{Bool}/a\}\phi$. Rather than allow b to become free, the solution is to bind it again in the result type.

Generalising from this example gives the following typing rule for applications, which is shown in Figure 2 and replaces (fapp) in Figure 1:

$$\text{(ifapp)}\ \frac{\Gamma \vdash r : \forall\Delta.\sigma \to \phi \quad \Gamma \vdash u : \psi}{\Gamma \vdash r\ u : \forall(\Delta\backslash\sigma).\{\psi/[\Delta]\,\sigma\}\phi}\mathsf{FTV}(\psi) \cap (\Delta\backslash\sigma) = \emptyset$$

Here $\Delta\backslash\sigma$ is the sub-sequence of Δ consisting of those variables not free in σ; these are the quantified variables that are not fixed by u. Note that (ifapp) only applies if the match exists, which is not always the case; for example there is no match of `Int` against `Bool`.

To illustrate (ifapp) in action, suppose `x` : `Int` and `y` : `Bool`. Then here are some terms and their types:

$$\begin{array}{rll}
\texttt{Pair Int Bool} : \texttt{Int} \to \texttt{Bool} \to (\texttt{Int}, \texttt{Bool}) & & (1) \\
\texttt{Pair Int Bool x} : \texttt{Bool} \to (\texttt{Int}, \texttt{Bool}) & & (2) \\
\texttt{Pair x} : \forall b.b \to (\texttt{Int}, b) & & (3) \\
\texttt{Pair x Bool} : \texttt{Bool} \to (\texttt{Int}, \texttt{Bool}) & & (4) \\
\texttt{Pair x Bool y} : (\texttt{Int}, \texttt{Bool}) & & (5) \\
\texttt{Pair x y} : (\texttt{Int}, \texttt{Bool}) & & (6)
\end{array}$$

The first two examples are well-typed System **F** as well as System **IF**, with the expected types; you do not *have* to drop type applications in System **IF**! Example (3) is more interesting; here the value argument `x` instantiates one, but only one, of the type variables in `Pair`'s type, leaving a term quantified in just one type variable. Incidentally, it would have made no difference if the type of `Pair` had been quantified in the other order ($\forall b, a.a \to b \to (a, b)$): example (3) would still have the same type. Examples (4–6) illustrate that the polymorphic function `(Pair x)` can be applied to a type (examples (4,5)) or to a term (example (6)).

Like **F**, System **IF** is robust to program transformation. For example, suppose $f : \forall \Delta.\sigma_1 \to \sigma_2 \to \phi$, and $g : \forall \Delta.\sigma_2 \to \sigma_1 \to \phi$, so that their types differ only the order of the two arguments. Then if $(f\ t_1\ t_2)$ is well-typed, so is $(g\ t_2\ t_1)$. This property relies on the ability of (ifapp) to re-abstract the variables not bound by the match. In particular, suppose $f : \forall a.a \to \texttt{Int} \to a$, $g : \forall a.\texttt{Int} \to a \to a$. Now consider the application $(f\ \texttt{True}\ \texttt{3})$. The partial application $(f\ \texttt{True})$ will instantiate a to `Bool`, yielding a result of type $\texttt{Int} \to \texttt{Bool}$ which is then applied to `3`. Now consider the arguments in the other order, in $(g\ \texttt{3}\ \texttt{True})$. The partial application $(g\ \texttt{3})$ yields a match that does *not* bind a, so the result is re-abstracted over a to give $\forall a.a \to a$. This function can be applied to `True` straightforwardly.

3.2 Dynamic semantics of System IF

Since (ifapp) admits more terms than (fapp), we need more reductions, too. In particular, it is necessary to reduce terms of the form $(\Lambda a.s)\ u$, where a type-lambda abstraction is applied to a term. A minimum requirement is that reduction should support the following generalisation of $(\beta 1)$ to handle the type abstractions:

$$(\beta 1')\ (\Lambda \Delta.\lambda x^{\sigma}.t)\ u^{\psi} \longrightarrow \Lambda(\Delta \backslash \sigma).\{u/x\}\{\psi/[\Delta]\sigma\}t \quad \text{if } \mathsf{FTV}(\psi) \cap (\Delta \backslash \sigma) = \emptyset.$$

This reduction matches (ifapp) closely. First it finds the match $\{\psi/[\Delta]\sigma\}$ that describes how to instantiate the type variables Δ (or rather, those that appear free in σ). Then it applies this substitution to t, and re-abstracts over the remaining type variables of Δ. The side condition, which can always be made to hold by α-conversion, simply ensures that the new abstraction does not capture any variables in u.

Notationally, this reduction makes use of Theorem 1, extended to System **IF**, which says that every term has a unique type. In the left-hand side of the reduction we write this type as a superscript on the *term*, thus u^{ψ}, although that is not part of the syntax of System **IF**. We could instead re-state the reduction in a way that is more faithful to the operational reality like this:

$$(\beta 1')\ (\Lambda \Delta.\lambda x^{\sigma}.t)\ u \longrightarrow \Lambda(\Delta \backslash \sigma).\{u/x\}\{\psi/[\Delta]\sigma\}t \quad \text{if } \mathsf{FTV}(\psi) \cap (\Delta \backslash \sigma) = \emptyset \text{ and } \exists \Gamma.\ \Gamma \vdash u : \psi.$$

Now the left-hand side can be matched purely structurally, while ψ is fixed by the new condition on the right-hand side. In operational terms, there is an easy algorithm to find ψ given a (well-typed) term u.

Although it is essential that the left-hand side of $(\beta 1')$ reduces to its right hand side, it is rather unsatisfactory as a one-step reduction *rule*. From a practical perspective, it eliminates a string of lambdas all at once, and may require reduction under a big lambda to bring the left-hand side to the required form.

From a theoretical perspective, it will prove to be a consequence of other rules to be introduced now.

The whole point of System **IF** is that one can omit many type applications, but this is not intended to express or create new meanings: when $(r\ u)$ and $(r\ \psi\ u)$ have the same meaning they should have the same normal form. For example, with the rules we have so far, these two terms would be distinct normal forms:

$$(\lambda x^{\forall a.a \to a}.\, x\ \texttt{Bool}\ \texttt{True}) \quad \text{and} \quad (\lambda x^{\forall a.a \to a}.\, x\ \texttt{True}).$$

Their equality is achieved by adding reduction rules ($\xi 1$) and ($\xi 2$) (they also appear in Figure 2):

$$\begin{array}{ll} (\xi 1)\ r^{\forall a, \Delta.\sigma \to \phi}\ \psi\ u \longrightarrow r\ u & \text{if } a \in \mathsf{FTV}(\sigma) \backslash \Delta \\ (\xi 2)\ r^{\forall a, \Delta.\sigma \to \phi}\ \psi\ u \longrightarrow r\ u\ \psi & \text{if } a \notin \mathsf{FTV}(\sigma) \backslash \Delta \end{array}$$

The first eliminates a type application altogether when the immediately following term argument fixes the instantiation of the polymorphic function – that is, when a is mentioned in σ[1]. For example, if $\texttt{Leaf} : \forall a.a \to \texttt{Tree}\ a$ then

$$\texttt{Leaf}\ \psi\ u^{\psi} \longrightarrow \texttt{Leaf}\ u^{\psi}.$$

By itself, this rule is not enough, as can be seen in $\texttt{Pair}\ \psi\ \tau\ s\ t$ since the type of s does not determine the value of τ which is instead given by t. This situation is handled by rule ($\xi 2$).

These two reductions may also be regarded as optimisation rules, that can be applied statically to remove redundant type applications from an System **IF** program. For example:

$$\begin{array}{rl} \texttt{Pair}\ \psi\ \tau\ s\ t \longrightarrow & \texttt{Pair}\ \psi\ s\ \tau\ t \text{ by } (\xi 2) \\ \longrightarrow & \texttt{Pair}\ s\ \tau\ t \quad \text{by } (\xi 1) \\ \longrightarrow & \texttt{Pair}\ s\ t \quad\ \ \text{by } (\xi 1) \end{array}$$

as desired. Here is another example. Suppose $\texttt{Inl} : \forall a, b.a \to a + b$, and $\texttt{Inr} : \forall a.\forall b.b \to a + b$. Then

$$\begin{array}{rl} \texttt{Inl Int Bool 3} \longrightarrow & \texttt{Inl Int 3 Bool} \text{ by } (\xi 2) \\ \longrightarrow & \texttt{Inl 3 Bool} \quad\ \ \text{by } (\xi 1) \end{array}$$

$$\begin{array}{rl} \texttt{Inr Int Bool True} \longrightarrow & \texttt{Inr Int True} \quad \text{by } (\xi 1) \\ \longrightarrow & \texttt{Inr True Int} \quad \text{by } (\xi 2) \end{array}$$

In these two examples, notice that one type application necessarily remains, because the argument of `Inl` and `Inr` only fixes one of the type parameters. (At least, it necessarily remains if `Inr` is regarded as a constant; if it is replaced by its Church encoding then further reductions can take place.) Exactly the same thing happens with the `Nil` of the list type.

[1] Technically we need $a \in \mathsf{FTV}(\sigma) \backslash \Delta$, since nothing prevents a appearing in Δ.

Note that ($\xi 1$) and ($\xi 2$) overlap with the rule ($\beta 2$) since a term of the form $(\Lambda a.t)\ \psi\ u$ can sometimes be reduced to both $\{\psi/a\}t\ u$ (by $\beta 2$) and $(\Lambda a.t)\ u$ (by $\xi 1$). Now, if t is λ-abstraction $\lambda x.s$ then both sides reduce to $\{u/x\}\{\psi/a\}s$ by ($\beta 1'$). However, if t and u are variables then $(\Lambda a.t)\ u$ is irreducible by the existing rules so that confluence would fail. That is, ($\beta 1'$) is just too coarse.

The solution is to add rules that act on terms of the form $(\Lambda a.t)\ u$ namely:

$$
\begin{array}{lll}
(\mu 1)\ (\Lambda a.t^{\forall \Delta.\sigma \to \phi})\ u^{\psi} \longrightarrow \{\psi/[a,\Delta]\sigma\}t\ u & & \text{if } a \in \mathsf{FTV}(\sigma)\setminus\Delta \\
 & & \text{and } \mathsf{FTV}(t)\cap\Delta = \emptyset \\
(\mu 2)\ (\Lambda a.t^{\forall \Delta.\sigma \to \phi})\ u^{\psi} \longrightarrow \Lambda a.(t\ u) & & \text{if } a \notin \mathsf{FTV}(u)\cup\mathsf{FTV}(\sigma)\setminus\Delta
\end{array}
$$

Rule ($\mu 1$) is akin to ($\beta 2$), in that it substitutes for the type variable a in the body t. Unlike System **F**, however, the type to substitute is not explicit; instead, it is found by matching the type ψ of the value argument u against the argument type σ of t. This match yields the substitution $\{\psi/[a,\Delta]\sigma\}$, which will certainly bind a, but also may (and perhaps must) bind variables in Δ. The side condition $\mathsf{FTV}(t)\cap\Delta = \emptyset$, which can always be made true by α-conversion, ensures that the substitution does not accidentally instantiate an unrelated free type variable of t.

The rule ($\mu 2$) is more straightforward. It simply commutes the abstraction and application when they do not interfere. Note that the side condition $a \notin \mathsf{FTV}(u)$ can be made true by α-conversion. Now ($\beta 1'$) is a consequence of ($\beta 1$) and ($\mu 1$) and ($\mu 2$) so the full set of reduction rules is given by the β-rules of System **F** plus the four new rules in Figure 2.

3.3 Translation to System F

This subsection formalises the close relationship between System **IF** and System **F**.

Theorem 3. *The embedding of System* **F** *into System* **IF** *preserves typing, type erasure and reduction.*

Note that some normal forms of System **F** are reducible in System **IF**. For example, it may happen that a normal form $x\ \psi\ u$ of System **F** reduces to $x\ u$ by ($\xi 1$).

$$
\begin{array}{rl}
[\![x^{\sigma}]\!] &= x^{\sigma} \\
[\![r^{\forall \Delta.\sigma \to \phi}\ u^{\psi}]\!] &= \Lambda(\Delta\setminus\sigma).[\![r]\!]\ \{\psi/[\Delta]\sigma\}\Delta\ [\![u]\!] \\
[\![\lambda x^{\sigma}.s]\!] &= \lambda x^{\sigma}.[\![s]\!] \\
[\![r\ \psi]\!] &= [\![r]\!]\ \psi \\
[\![\Lambda a.s]\!] &= \Lambda a.[\![s]\!]
\end{array}
$$

Fig. 3: Translation from System **IF** to System **F**

This natural embedding is complemented by a translation in the opposite direction, from System **IF** into System **F**, that makes the implicit type applications explicit. The translation is shown in Figure 3. Although it is fairly well behaved, it does not preserve the reduction rule ($\xi 2$). Rather, the left-hand and right-hand sides of ($\xi 2$) translate repectively to

$$\Lambda a.\Lambda(\Delta\backslash\sigma).[\![r]\!]\ \psi\ \Delta'\ [\![u]\!] \qquad \text{and} \qquad (\Lambda a.\Lambda(\Delta\backslash\sigma).[\![r]\!]\ a\ \Delta'\ [\![u]\!])\ \psi$$

for some Δ'. Here, the right-hand side reduces to the left-hand side by ($\beta 2$). In this case the permutation of the arguments has caused the direction of reduction to be reversed. However, although the translation does not preserve reduction, it does preserve equality, the symmetrical relation generated by the rules.

The same type erasure mechanism used for System **F** can be applied to terms of System **IF** to yield pure λ-terms.

Theorem 4. *There is a translation $[\![-]\!]$ from terms of System* **IF** *to those of System* **F** *that preserves typing, type erasure and equality of terms. Further, if t is a term in System* **F** *then $[\![t]\!]$ is t itself.*

Proof. That the translation in Figure 3 preserves type derivations follows by induction on the structure of the derivation. That it preserves type erasure is immediate. The preservation of terms in System **F** follows by induction on the structure of the term. The only non-trivial case concerns an application $r\ u$ but then the type of r cannot bind any type variables so that the translation becomes

$$[\![r^{\sigma\to\phi}\ u^{\sigma}]\!] = [\![r]\!]\ [\![u]\!].$$

Now consider equality. It is enough to show that when both sides of each reduction rule are translated then they are either equal, or have a common reduct. The rule ($\beta 1$) is preserved since $[\![(\lambda x^{\sigma}.s)\ u]\!] = (\lambda x^{\sigma}.[\![s]\!])\ [\![u]\!] \longrightarrow \{[\![u]\!]/x\}[\![s]\!]\} = [\![\{u/x\}s]\!]$ where the last equation is a trivial lemma. A similar argument applies to ($\beta 2$). Both sides of the rule ($\xi 1$) have the same translation. The argument for ($\xi 2$) is already given. The translation of ($\mu 1$) is given by the ($\beta 2$) reduction

$$\Lambda(\Delta\backslash\sigma).(\Lambda a.[\![t]\!])\ \theta a\ \ \theta\Delta\ [\![u]\!] \longrightarrow \{\theta a/a\}(\Lambda(\Delta\backslash\sigma).[\![t]\!]\ \theta\Delta\ [\![u]\!])$$

where θ is $\{\psi/[a,\Delta]\sigma\}$. The translation of ($\mu 2$) is given by the ($\beta 2$) reduction

$$\Lambda a.\Lambda(\Delta\backslash\sigma).(\Lambda a.[\![t]\!])\ a\ \ \theta\Delta\ [\![u]\!] \longrightarrow \Lambda a.\Lambda(\Delta\backslash\sigma).[\![t]\!]\ \theta\Delta\ [\![u]\!]$$

where θ is $\{\psi/[a,\Delta]\sigma\}$ which is also $\{\psi/[\Delta]\sigma\}$ since $a \notin \mathsf{FTV}(\sigma)$.

This completes the proof that the translation preserves the equality generated by rewriting.

□

3.4 Properties of System IF

System **IF** is a little more complicated than System **F**, but they share many of the same good properties, including being strongly normalising and confluent. Notably, it shares the unique-type property described in Section 2.2, as can be seen by inspection of the typing rules.

Lemma 1 (Substitution Lemma).

1. *If there is a derivation of $t : \sigma$ and θ is a type substitution then there is a derivation of $\theta t : \theta\sigma$.*
2. *If $s : \phi$ is a term and x^σ is a variable that may be free in s and $u : \sigma$ is a term then there is a derivation of $\{u/x\}s : \phi$.*

Proof. The proofs are by straightforward induction on the structure of the terms. □

Theorem 5. *Reduction in System* **IF** *preserves typing.*

Proof. The proof is by induction on the structure of the reduction. Without loss of generality, the reduction is a rule. If it is either ($\beta 1$) or ($\beta 2$) then apply the Substitution Lemma.

If the rule is ($\xi 1$) with $u : \tau$ then $r\ \psi : \{\psi/a\}(\forall\Delta.\sigma \to \phi)$ and so $r\ \psi\ u : \forall(\Delta\backslash\sigma).\{\tau/[\Delta]\{\psi/a\}\sigma\}\{\psi/a\}\phi$ while

$$r\ u : \forall(a, \Delta\backslash\sigma).\{\tau/[a, \Delta]\sigma\}\phi.$$

Now $a \in \mathsf{FTV}(\sigma)$ implies that $\{\tau/[a, \Delta]\sigma\} = \{\tau/[\Delta]\{\psi/a\}\sigma\} \circ \{\psi/a\}$ (where $\circ$ denotes composition of substitutions). This yields the result.

If the rule is ($\xi 2$) with $u : \tau$ then $r\ \psi : \{\psi/a\}(\forall\Delta.\sigma \to \phi)$ whose type is also $\forall\Delta.\sigma \to \{\psi/a\}\phi)$ since a is not free in σ. Hence $r\ \psi\ u$ has type $\forall(\Delta \setminus \sigma).\{\tau/[\Delta]\sigma\}\{\psi/a\}\phi$ which is the type $\forall(\Delta\backslash\sigma).\{\psi/a\}\{\tau/[\Delta]\sigma\}\phi$ of $r\ u\ \psi$.

If the rule is ($\mu 1$) then the left-hand side has type $\forall(\Delta\backslash\sigma).\upsilon\phi$ where $\upsilon = \{\psi/[a, \Delta]\sigma\}$. Also, the right-hand side has type

$$\forall(\Delta\backslash\sigma).\{\psi/[\Delta]\{\upsilon a/a\}\sigma\}\{\upsilon a/a\}\phi$$

which is the same since $\upsilon = \{\psi/[\Delta]\{\upsilon a/a\}\sigma\} \circ \{\upsilon a/a\}$.

If the rule is ($\mu 2$) then the left-hand side has type $\forall a.\forall(\Delta \setminus \sigma).\upsilon\phi$ where $\upsilon = \{\psi/[a, \Delta]\sigma\}$ is also $\{\psi/[\Delta]\sigma\}$ since a is not free in σ. Hence, the type is that of the right-hand side, too.

□

Theorem 6. *Type erasure maps ($\beta 1$) to β-reduction of the pure λ-calculus and maps all other rules to equations.*

Proof. The proof is immediate. □

Theorem 7. *Reduction in System* **IF** *is strongly normalising.*

Proof. Observe that if t is a term in System **F** then its type erasure is strongly normalising in the pure λ-calculus, since any reduction of the erasure is the image of some non-empty reduction sequence in System **F**. Since the translation from System **IF** to System **F** preserves the type erasure and ($\beta 1$) this property extends to all of System **IF**. Thus any reduction sequence in System **IF** contains finitely many instances of ($\beta 1$). Hence, to prove strong normalisation, it

suffices to consider reduction sequences without any uses of $(\beta 1)$. The remaining reduction rules all reduce the rank ρ of the terms, as defined by

$$\begin{aligned}\rho(x^{\sigma}) &= 0\\ \rho(r\ u) &= 2\rho(r) + \rho u\\ \rho(r\ U) &= \rho(r) + 1\\ \rho(\Lambda a.s) &= 2\rho(s) + 1.\end{aligned}$$

For example, $\rho(r\ \psi\ u) = 2(\rho(r)+1) + \rho(u) > 2\rho(r) + \rho(u) + 1 = \rho(r\ u\ \psi)$ and $\rho((\Lambda a.s)\ u) = 2(\rho(s)+1) + \rho(u) > 2\rho(s) + \rho(u) + 1 = \rho(\Lambda a.s\ u)$. The other three reduction rules are easily checked.

□

Theorem 8. *Reduction in System* **IF** *is Church-Rosser.*

Proof. Since reduction is strongly normalising, it is enough to prove that every critical pair can be resolved. Those which are already present in System **F** are resolved using its Church-Rosser property. The new reduction rules cannot overlap with $(\beta 1)$ for typing reasons. Nor can they overlap with each other. Hence the only remaining critical pairs involve $(\beta 2)$ and $(\xi 1)$ or $(\xi 2)$.

For $(\xi 1)$, a term of the form $(\Lambda a.s)\ \psi_1\ u^{\psi}$ rewrites to both $\{\psi_1/a\}s\ u$ and $(\Lambda a.s)\ u$. The latter term further reduces by $(\mu 1)$ to $\theta s\ u$ where θ is the restriction of $\{\psi/[a,\Delta]\sigma\}$ to a. Now this must map a to ψ_1 since a is free in σ and $\{\psi/[\Delta]\{\psi_1/a\}\sigma\}$ exists (from the typing of the original term). Hence $\{\psi_1/[a,\Delta]\sigma\}s\ u$ is exactly $\{\psi/a\}s\ u$.

For $(\xi 2)$ a term of the form $(\Lambda a.s)\ \psi_1\ u$ rewrites to both $\{\psi_1/a\}s\ u$ and $(\Lambda a.s)\ u\ \psi_1$. The latter term further reduces to $(\Lambda a.s\ u)\ \psi_1$ and thence to the former term. □

3.5 Extension to higher kinds

The side conditions for the novel reduction rules constrain the appearance of bound type variables in quantified types, and the matching process inspects their syntactic structure. Type safety requires that these properties be stable under substitution.

In System **IF**, like System **F**, this is automatically true. But what if one were to add functions at the type level, as is the case in $\mathbf{F}\omega$, for example? Then, b is free in the type $(a\ b)$, but if we were to substitute $\lambda x.\texttt{Int}$ for a, then $(a\ b)$ would reduce to `Int` in which b is no longer free. Similarly, computing the match $\{\texttt{List Int}/[b](a\ b)\}$, as defined in Figure 2, would yield the substitution $\{\texttt{Int}/b\}$; but if we were to substitute $\lambda x.x$ for a, the match would become $\{\texttt{List Int}/[b]b\}$, which yields quite a different result.

None of these problems arise, however, in the fragment of $\mathbf{F}\omega$ that is used by Haskell (Peyton Jones 2003). Haskell permits higher-kinded constants (introduced by data type declarations), but has no type-level lambda, and hence no reductions at the type level. In this setting, simple first-order matching suffices despite the use of higher kinds. For example, adding `List` as a constant of kind

$\star \to \star$ (with no reduction rules) causes no difficulty; indeed, this same property is crucial for type inference in the Haskell source language.

In short, System **IF** as presented above can readily be extended with type constants and type variables of higher kind, to serve as an intermediate language for Haskell. The syntax of types would then become

$$\begin{array}{c} S, T ::= \langle \text{type constructors} \rangle \\ \sigma, \tau, \phi, \psi ::= a \mid \sigma \to \sigma \mid \forall a.\sigma \mid T \mid \sigma\ \phi \end{array}$$

(note no lambda), together with kinding rules to ensure that types are well-kinded. We leave for future work the question of whether and how **IF** can be further extended to accommodate full type-level lambda.

3.6 Eta-rules

In pure λ-calculus, the (η)-equality

$$\lambda x.r\ x = r$$

reflects the intuition that everything is a function. It is of interest here for two reasons. One is to tighten the connection between System **IF** and System **F**. The other is to support its use in compiler optimisations.

Traditionally, (η) it has been expressed as a contraction (reducing $\lambda x.r\ x$ to r), which is appropriate in practice. Note, however, that for many purposes it is better thought of as an expansion (Jay and Ghani 1995; Ghani 1997). In System **F** the η-contraction rules are

$$\begin{array}{ll} (\eta 1)\ \lambda x.r\ x \longrightarrow r & \text{if } x \notin \mathsf{ftv}(r) \\ (\eta 2)\ \Lambda a.r\ a \longrightarrow r & \text{if } a \notin \mathsf{FTV}(r). \end{array}$$

In System **IF**, the rule $(\eta 1)$ must be made more general, to reflect the implicit action on type variables. Given a term $r : \forall \Delta.\sigma \to \phi$ and a variable $x : \sigma$ then $r\ x$ will instantiate the type variables in $\Delta \cap \mathsf{FTV}(\sigma)$ while leaving the rest bound. Hence $r\ x\ (\Delta \backslash \sigma) : \phi$ and so

$$\lambda x^{\sigma}.r\ x\ (\Delta \backslash \sigma) : \sigma \to \phi$$

has the same type as $r\ \Delta$. In this way, the rules become:

$$\begin{array}{ll} (\eta 1')\ \lambda x^{\sigma}.r^{\forall \Delta.\sigma \to \phi}\ x\ (\Delta \backslash \sigma) \longrightarrow r\ \Delta & \text{if } x \notin \mathsf{ftv}(r) \\ (\eta 2) \qquad\qquad \Lambda a.r\ a \longrightarrow r & \text{if } a \notin \mathsf{FTV}(r). \end{array}$$

Of course, if the redex of $(\eta 1')$ is well-typed in System **F** then Δ is empty and the standard rule emerges. When these η-contractions are added, the resulting systems are called System **F**η and System **IF**η respectively.

The rule $(\eta 1')$ is unlike all previous rules considered, in that it is not stable under substitution for type variables in Δ. If θ is a type substitution then the reduction relation defined by the rules must be broadened to include

$$\lambda x^{\theta\sigma}.r^{\forall \Delta.\sigma \to \phi}\ x\ \theta(\Delta \backslash \sigma) \longrightarrow r\ (\theta\Delta) \text{ if } x \notin \mathsf{ftv}(r).$$

To apply the rule in this form, it suffices to discover θ by type matching of σ against the given type of x etc.

The good properties established in the last two sub-sections continue to hold in the presence of the η-rules.

Theorem 9. *The translation $[\![-]\!]$ from terms of System* **IF** *to those of System* **F** *maps $(\eta 2)$ to $(\eta 2)$ and maps $(\eta 1')$ to a sequence of reductions using $(\eta 2)$ followed by $(\eta 1)$. Further, for each term t in System* **IF**η*, its translation $[\![t]\!]$ reduces to t in System* **IF**η*.*

Proof. The translation of the redex of $(\eta 1')$ is

$$\lambda x^{\sigma}.(\Lambda(\Delta\backslash\sigma).[\![r]\!]\ \Delta\ x)\ (\Delta\backslash\sigma)$$

which reduces by $(\eta 2)$ to $\lambda x.[\![r]\!]\ \Delta\ x$ and then to $[\![r]\!]\ \Delta$ by $(\eta 1)$. The translation of $(\eta 2)$ is $(\eta 2)$.

The proof that $[\![t]\!] \longrightarrow t$ is by induction on the structure of t. The only non-trivial case is an application $r\ u$. To its translation apply the ξ-rules to get a term of the form $\Lambda(\Delta\backslash\sigma).[\![r]\!]\ [\![u]\!]\ (\Delta\backslash\sigma)$ which reduces by $(\eta 2)$ to $[\![r]\!]\ [\![u]\!]$. In turn, this reduces to $r\ u$ by induction.

Theorem 10. *Reduction in System* **IF**η *is strongly normalising.*

Proof. The proof is as for System **IF**, but now relying on the translation to System **F**η.

Theorem 11. *Reduction in System* **IF**η *is Church-Rosser.*

Proof. It suffices to check the new critical pairs that arise from interaction with the η-rules. The critical pair involving $(\beta 2)$ and $(\eta 2)$ is resolved as in **F**. The other critical pairs are of $(\eta 1')$ with $(\beta 1), (\xi 1), (xi 2), (\mu 1)$ and $(\mu 2)$. Let us consider them in turn, in relation to the rule $(\eta 1')$.

If r is $\lambda x.s$ then Δ is empty and $\lambda x.(\lambda x.s)\ x$ reduces by $(\eta 1')$ and $(\beta 1)$ to $\lambda x.s$.

If r is $t\ a$ for some $t : \forall a.\forall\Delta.\sigma \to \phi$ then $\lambda x.t\ a\ x\ (\Delta\backslash\sigma)$ reduces to $t\ a\ \Delta$ by $(\eta 1')$. The result of applying either $(\xi 1)$ or $(\xi 2)$ to the original term can be further reduced by $(\eta 1')$ to $t\ a\ \Delta$. More generally, if r is some $t\ \psi$ then the applications of $(\eta 1')$ must have the substitution of ψ for a applied to them.

If r is $\Lambda a.s$ then $(\eta 1')$ reduces this to $(\Lambda a.s)\ a\ (\Delta\backslash\sigma)$. This in turn reduces to $s\ (\Delta\backslash\sigma)$. If $a \in \mathsf{FTV}(\sigma)\backslash\Delta$ then this is also the result of applying first $(\mu 1)$ and then $(\eta 1')$. If $a \notin \mathsf{FTV}(\sigma)\backslash\Delta$ then this is also the result of applying first $(\mu 2)$ and then $(\beta 2)$ and $(\eta 1')$.

4 Practical implications

Apart from its interest as a calculus in its own right, we were interested in applying System **IF** in a real compiler. We therefore modified the Glasgow Haskell

Compiler (GHC), a state-of-the-art compiler for Haskell, to use System **IF** as its intermediate language. This process turned out (as hoped) to be largely straightforward: the changes were highly localised, and only a few hundred lines of code in a 100,000-line compiler had to be modified in a non-trivial way.

GHC already uses a mild extension of System **F** as its intermediate language: at the type level it admits generalised algebraic data types (GADTs) and higher-kinded type variables; while in terms it allows recursive `let` bindings, data constructors (and applications thereof), and `case` expressions to decompose constructor applications. We extended System **IF** in an exactly analogous way, a process which presented no difficulties, although we do not formalise it here.

While GHC maintains full type information throughout its optimisation phases, it performs type erasure just before code generation, so types have no influence at run-time. The effect of using System **IF** instead of **F** is therefore confined to compile time.

4.1 Variable bindings and occurrences

In practical terms, the reader may wonder about the space implications of attaching a type to every variable *occurrence*, rather than attaching the variable's type only to its *binding* (Section 2.1). In fact, GHC already does exactly this, reducing the additional space costs (compared to annotating only the binding sites) to zero by sharing. The data type that GHC uses to represent terms looks like this:

```
data CoreExpr = Var Id | Lam Id CoreExpr | ...
data Id = MkId Name Type
```

An `Id` is a pair of a `Name` (roughly, a string), and its `Type`, so it represents the formal notation x^σ. An `Id` is used both at the *binding site* of the variable (constructor `Lam`) and at its *occurrences* (`Var`). GHC is careful to ensure that every occurrence shares the same `Id` node that is used at the binding site; in effect, all the occurrences point to the binder. This invariant is maintained by optimisation passes, and by operations such as substitution.

As a consequence, like System **F** itself, every term in GHC's intermediate language has a unique type, independent of its context (Section 2.2). That is, we can (and GHC does) provide a function `exprType` that extracts the type of any term:

```
exprType :: CoreExpr -> Type
```

Notice that `exprType` does not require an environment; it can synthesise the type of a term by inspecting the term alone, in a single, bottom-up traversal of the term. (Here, we assume that the term is well-typed; `exprType` is not a type-checking algorithm.) This function `exprType` makes concrete the idea that every term has a unique type (Section 2.1), and it is extremely valuable inside GHC, which is why maintaining the unique-type property was a key requirement of our design.

Module	GHC Core		System **IF**		Reduction(%)	
	Size	Type apps	Size	Type apps	Size	Type apps
Data.Tuple	169,641	10,274	131,349	0	23%	100%
Data.Generics.Instances	36,038	1,774	32,488	578	10%	67%
Data.Array.Base	32,068	2,498	26,468	397	17%	84%
Data.Sequence	29,468	2,124	24,532	354	17%	83%
Data.Map	21,217	1,566	17,958	334	15%	79%
Data.Array.Diff	16,286	895	14,067	73	14%	92%
GHC.Float	16,100	414	15,353	50	5%	88%
Data.IntMap	14,614	1,025	12,363	209	15%	80%
System.Time	14,499	914	12,934	338	11%	63%
GHC.Read	14,210	959	11,110	141	22%	85%
GHC.Real	14,094	355	13,400	54	5%	85%
GHC.IOBase	13,698	711	11,494	212	16%	70%
GHC.Handle	13,504	902	11,938	192	12%	79%
GHC.Arr	13,455	837	11,490	49	15%	94%
Data.ByteString	12,732	1,104	10,632	230	16%	79%
Foreign.C.Types	12,633	359	11,987	48	5%	87%
... and 114 other modules ...						
TOTAL	792,295	46,735	676,615	7,257	15%	84%
TOTAL (omitting Data.Tuple)	622,654	36,461	545,266	7,257	12%	80%

Fig. 4: Effect of using System **IF**

4.2 Optimisation and transformation

One might wonder whether, in changing from System **F** to **IF**, we had to rewrite every transformation or optimisation in the compiler. Happily, we did not. Almost all transformations now rewrite **IF** to **IF** without change, and certainly without introducing and re-eliminating the omitted type applications.

The non-trivial changes were as follows:

- The new typing rule (ifapp) amounts to extending the `exprType` function, and Core Lint. The latter is a type checker for GHC's intermediate language, used only as a consistency check, to ensure that the optimised program is still well typed. If the optimisations are correct, the check will always succeed, but it is extremely effective at finding bugs in optimisations.
  It turned out to be worth implementing the type-matching function (Section 3.1) twice. For Core Lint the full matching algorithm is required, but `exprType` operates *under the assumption that the term is well typed*. In the latter case several short-cuts are available: there is no need to check that matching binds a variable consistently at all its occurrences, kind-checks (usually necessary in GHC's higher-kinded setting) can be omitted, and matching can stop as soon as all the quantified variables have been bound.
- The new reduction rules $(\xi 1), (\xi 2), (\mu 1), (\mu 2), (\eta 1')$ all appear as new optimising transformations. GHC's optimiser goes to considerable lengths to ensure

that transformations "cascade" well, so that many transformations can be appplied successively in a single, efficient pass over the program. Adding the new rules while retaining this property was trickier than we expected.
- Eta-reduction, one of GHC's existing transformations, becomes somewhat more complicated (Section 3.6).
- There is one place that that we *do* reconstruct the omitted type arguments, namely when simplifying a `case` that scrutinises a constructor application, where the constructor captures existential variables. For example, consider the following data type:

```
data T where { MkT :: forall a. a -> (a->Int) -> T }
```

Now suppose we want to simplify the term

```
case (MkT 'c' ord) of
  MkT a (x:a) (f:a->Int) -> ...
```

(Here `ord` has type `Char->Int`.) Then we must figure out that the existential type variable `a`, bound by the pattern, should be instantiated to `Char`, which in turn means that we must re-discover the omitted type argument of the `MkT` constructor.

4.3 Code size

We compiled all 130 modules of the `base` library packages, consisting of some 110,000 lines of code. Each module was compiled to GHC's Core language, which is a variant of System **F**. We wrote a special pass to eliminate redundant type applications by applying rules ($\xi 1$) and ($\xi 2$) exhaustively, and measured the size of the program before and after this transformation. This "size" counts the number of nodes in the syntax tree of the program including the sizes of types, except the types at variable occurrences since they are always shared. Apart from the sharing of a variable's binding and its occurrences we do not assume any other sharing of types (see Section 5.2). We also counted the number of type applications before and after the transformation.

The results are shown in Figure 4, for the largest 16 modules, although the total is taken over all 130 modules. The total size of all these modules taken together was reduced by around 15%, eliminating nearly 84% of all type applications. These figures are slightly skewed by one module, `Data.Tuple`, which consists entirely of code to manipulate tuples of various sizes, and which is both very large and very uncharacteristic (all of its type applications are removed). Hence, the table also gives the totals excluding that module; the figures are still very promising, with 80% of type applications removed.

Manual inspection shows that the remaining type applications consist almost exclusively of

- Data constructors where the arguments do not fully specify the result type (such as `Nil`).

– Calls to Haskell's `error` function, whose type is

$$\texttt{error} : \forall a.\texttt{String} \to a$$

There are a handful of other places where type applications are retained. For example, given `map` and `reverse` with their usual types, and `xs` : `List Int`, then in the term

```
(map (reverse Int) xs)
```

the type application `(reverse Int)` cannot be eliminated by our rules. In practice, however, data constructors and error calls dominate, and that is fair enough, because the type applications really are necessary if every term is to have unique, synthesisable type.

4.4 Compilation time

The proof of the pudding is in the eating. System **IF** has smaller terms, but its reduction rules are more complicated, and computing the type of a term involves matching prior to instantiation. Furthermore, although fewer types appear explicitly in terms, some of these omitted types might in practice be constructed on-the-fly during compilation, so it is not clear whether the space saving will translate into a time saving.

We hoped to see a consistent improvement in the execution time of the compiler, but the results so far are disappointing. We measured the total number of bytes allocated by the compiler (a repeatable proxy for compiler run-time) when compiling the same 130 modules as Section 4.3. Overall, allocation decreased by a mere 0.1%. The largest reduction was 4%, and the largest increase was 12%, but 120 of the 130 modules showed a change of less than 1%. Presumably, the reduction in work that arises from smaller types is balanced by the additional overheads of System **IF**.

On this evidence, the additional complexity introduced by the new reduction rules does not pay its way. Nevertheless, these are matters that are dominated by nitty-gritty representation details, and the balance might well be different in another compiler.

5 Related work

5.1 Type inference

In source languages with type inference, such as ML or Haskell, programmers *never* write type applications — instead, the type inference system infers them. The classic example of such a system is the Hindley-Milner type system (Milner 1978), which has been hugely influential; but there are many other type inference systems that allow a richer class of programs than Hindley-Milner. For example, Pierce and Turner's Local Type Inference combines type information from arguments to choose the type instantiation for a function, in the context of a

language with subtyping (Pierce and Turner 1998). Their paper also introduced the idea of *bidirectional* type inference, which Peyton Jones *et al* subsequently applied in the context of Haskell to improve type inference for higher-rank types (Peyton Jones et al. 2007).

Stretching the envelope even further, Le Botlan and Rémy's ML^F language supports *impredicative* polymorphism, in which a polymorphic function can be called at a polytype (Le Botlan and Rémy 2003). Pfenning goes further still, describing the problem of *partial type inference* for full $F\omega$, including lambda at the type level (Pfenning 1988). Type abstractions ($\Lambda a.t$) are retained, but type annotations may be omitted from term abstractions (thus $\lambda x.t$ rather than $\lambda x^{\sigma}.t$), and type applications may be abbreviated to mere placeholders (thus $t\ [\,]$ instead of $t\ \sigma$). However, type inference in this system requires higher-order unification, and it lacks the unique-type property. Even more general use of such placeholders can be found in dependent type systems, which may be used to compress the size of proof-carrying code (Necula and Lee 1998) or to support dependently-typed programming languages (Norell 2007).

Such inference engines are invariably designed for *source* languages, and are less well suited for use as a calculus, or as a compiler intermediate language.

- Most type inference systems are less expressive than System **F**. For example, in Hindley-Milner, function arguments always have a monomorphic type; in many other systems types must be of limited rank, and the system is usually predicative.
  This lack of expressiveness matters, because the compiler may use a richer type system internally than is directly exposed to the programmer. Examples include closure conversion (Minamide et al. 1996), and the dictionary-passing translation for type classes (Wadler and Blott 1989).
- Type inference can be regarded as a constraint-solving problem, where the constraints are gathered non-locally from the program text, and the solver may be somewhat costly to run. In contrast, in a compiler intermediate language one needs to answer the question "what is the type of this subterm?", and to do so cheaply, and using locally-available information. For this purpose, the unique-type property of Section 2.2 is extremely helpful, but it is rare indeed for a system based on type inference to possess it.
- The restrictions that allow type inference are never fully robust to program transformation, and (in every case except the least expressive, Hindley-Milner) require *ad hoc* type annotations. For example, even if t is well-typed in a particular environment, $(\lambda f.t)\ f$ may not be, because, say, f's type may not be a monotype. Compilers perform inlining and abstraction all the time. Another way to make the same point is this: type inference systems usually treat a *fixed* source program text; they are never thought of as a *calculus* equipped with a type-preserving reduction semantics.

In short, type inference systems focus on programmers, whereas our focus is on compiler writers and logicians. Nevertheless, to the extent that one might regard System **F** as a language for programmers, we believe that **IF** should serve the same role as well or better.

5.2 Reducing the cost of types

A handful of papers address the question of the overheads of type information in type-preserving compilers. The most popular approach is to exploit common sub-structure by *sharing* common types or parts of types. These techniques come in two varieties: ones that *implicitly* share the representation of terms, and ones that express sharing *explicitly* in the syntax of terms.

For example, Petersen (Petersen 2005, Section 9.6) and Murphy (Murphy 2002) describe several approaches to type compression in the TILT compiler. Most of these are representation techniques, such as hash-consing and de-Bruijn representation, that can be used to implement type operations more efficiently. Shao *et al* devote a whole paper to the same subject, in the context of their FLINT compiler (Shao et al. 1998). Their techniques are exclusively of the implementation variety: hash-consing, memoisation, and advanced lambda encoding.

Working at the representation level is tricky, and seems to be sensitive to the context. For example Murphy reports a slow-down from using hash-consing, whereas Shao *et al* report a significant speed-up, and Petersen found that without hash-consing type-checking even a small program exhausted the memory on a 1Gbyte machine. (These differences are almost certainly due to the other techniques deployed at the same time, as we mention shortly.) Another reason that representation-level sharing is tricky is that it is not enough for two types to share memory; the sharing must also be *observable*. Consider, say, substitution. Unless sharing is observable, the substitution will happen once for each identical copy, and the results will be laboriously re-hash-consed together. Memory may be saved, but time is not.

A complementary approach, and the one that we discuss in this paper, is to change the intermediate language itself to represent programs more efficiently. For example, TILT has a `lettype` construct which provides for explicit sharing in type expressions. For example, using `lettype` we could write the nested `Pair` example from Section 2.3 like this:

$$
\begin{array}{l}
\texttt{lettype}\ a_1 = \sigma_1; \ldots; a_5 = \sigma_5 \\
\qquad\quad b_1 = (a_1, a_2) \\
\qquad\quad b_2 = (a_3, a_4) \\
\qquad\quad b_3 = (b_2, a_5) \\
\texttt{in Pair}\ b_1\ b_3 \\
\qquad\quad (\texttt{Pair}\ a_1\ a_2\ e_1\ e_2) \\
\qquad\quad (\texttt{Pair}\ b_2\ a_5\ (\texttt{Pair}\ a_3\ a_4\ e_3\ e_4)\ e_5).
\end{array}
$$

Such an approach carries its own costs, notably that type equivalence is modulo the environment of `lettype` bindings, but since TILT has a very rich notion of type equivalence anyway including full β-reduction in types, the extra pain is minimal. The gain appears to be substantial: FLINT does not have `lettype`, and Murphy suggests that this may be the reason that FLINT gets a bigger relative gain from hash-consing — `lettype` has already embodied that gain in TILT by construction.

Another example of changing the intermediate language to reduce type information is Chilpala *et al*'s work on strict bidirectional type checking (Chilpala et al. 2005). Their main goal is to drop the type annotation from a binder, for example writing $\lambda x.t$ instead of $\lambda x^{\sigma}.t$. It is possible to do this when the occurrence(s) of x completely fix its type. The paper only describes a simply-typed language, whereas our focus is exclusively on the type applications that arise in a polymorphic language. Furthermore, our approach relies on every term having a unique type, whereas theirs relies on inferring the unique types of the free variables of a term, starting from the type of the term itself. It remains to be seen whether the two can be combined, or which is more fruitful in practice.

Our focus is exclusively on reducing the cost of *compile-time* type manipulation. Other related work focuses on the cost of *run-time* type manipulation, for systems that (unlike GHC) do run-time type passing (Tolmach 1994; Saha and Shao 1998).

5.3 Pattern calculus

The approach to type quantification developed in this paper was discovered while trying to type pattern-matching functions (Jay and Kesner 2006) in which each case may have a different (but compatible) polymorphic type (Jay 2006). For example, consider a function $\texttt{toString} : \forall a.a \to \texttt{String}$ which is to have special cases for integers, floats, pairs, lists, etc. A natural approach is to define special cases for each type setting, as follows:

$$\begin{aligned}
\texttt{toStringInt} &: \texttt{Int} \to \texttt{String} \\
\texttt{toStringFloat} &: \texttt{Float} \to \texttt{String} \\
\texttt{toStringPair} &: \forall b, c.(b, c) \to \texttt{String} \\
\texttt{toStringList} &: \forall a.\texttt{List}\ a \to \texttt{String}
\end{aligned}$$

and then try to combine them into a single function. Then the application $\texttt{toString}\ (\texttt{List}\ a)$ could reduce to $\texttt{toStringList}\ a$ and $\texttt{toString}\ (b, c)$ could reduce to $\texttt{toStringPair}\ b\ c$ etc. in a form of `typecase`. However, this makes types central to reduction so that they cannot be erased. By making type applications implicit, it is possible to let the choice of special case be determined by the structure of the function argument instead of the type, as in

$$\begin{aligned}
\texttt{toString}\ 3 &\longrightarrow \texttt{toStringInt}\ 3 \\
\texttt{toString}\ 4.4 &\longrightarrow \texttt{toStringFloat}\ 4.4 \\
\texttt{toString}\ (\texttt{Pair}\ x\ y) &\longrightarrow \texttt{toStringPair}\ (\texttt{Pair}\ x\ y) \\
\texttt{toString}\ \texttt{Nil} &\longrightarrow \texttt{toStringList}\ \texttt{Nil}.
\end{aligned}$$

The proper development of this approach is beyond the scope of this paper; the point here is that implicit type quantification is the natural underpinning for this approach. Note, too, that there are natural parallels with object-orientation, where the object determines how to specialise the methods it invokes.

6 Further work

The focus of this paper is on removing redundant type applications but type applications are not the only source of redundant type information. For example, as (Chilpala et al. 2005) point out, non-recursive `let` expressions are both common (especially in A-normalised code) and highly redundant; in the expression (`let` x^{σ}`=`r `in` t) the type annotation on x is redundant since it can be synthesised from r. A similar point can be made for `case` expressions; for example, consider the expression

$$\lambda x^{\sigma}.\,\texttt{case}\ x\ \texttt{of}\ (p^{\phi}, q^{\psi})\texttt{->}\ t$$

Here, the type annotations on p and q are redundant, since they can be synthesised from the type of x.

One approach is to follow (Chilpala et al. 2005) by dropping these readily-synthesisable types at the language level. But nothing is gained from dropping type annotations on binders unless we also drop the annotations on *occurrences*, and that in turn loses the highly-desirable property that every term has a synthesisable type. An alternative, and perhaps more promising, approach is to work at the representation level, by regarding the type on such a binder simply as a cached or memoised call to `exprType`. In this way, if the type of the right-hand side of a non-recursive `let` binding was very large, the chances are that much sub-structure of that large type would be shared with variables free in that term. We have no data to back up these speculations, but it would be interesting to try.

So far we have assumed that the back end of the compiler performs type erasure, so that there is no run-time type passing. However, suppose one wants run-time type passing, to support `typecase` or reflection. It is immediately obvious how to compile System **F** to machine code, and still support run-time type passing — just make all type arguments into value arguments — but matters are not so obvious for System **IF**. This is an area for future work.

System **IF** contains a mixture of implicit and explicit type applications. This is extremely useful for backwards compatibility with System **F** but the resulting calculus is hardly minimal. Hence, there are a number of other possibilities for handling type applications. In particular, one can insist that all such are implicit, in a calculus of *quantified function types* (Jay 2006) whose types are given by

$$\sigma ::= a \ \mid\ [\Delta]\sigma \rightarrow \sigma$$

where the well-formed quantified function types $[\Delta]\sigma \rightarrow \phi$ play the same role as the type $\forall\Delta.\sigma \rightarrow \phi$ in System **F** but come with a guarantee that all type applications can be made implicit.

7 Conclusions

The formal beauty of System **F** together with the practical success of the Hindley-Milner type system have combined to set a high standard for anyone attempting

to improve in either direction. For studying the concept of parametric polymorphism System **F** appears ideal, despite its redundancies. For avoiding types, the Hindley-Milner system is spectacularly successful. The key observation of this paper is that one can eliminate much of the redundant type information in System **F** without sacrificing any expressive power or basic properties. The price of this approach is the need to add some redundancy to the reduction rules, so that there are several ways to reduce a type application.

System **IF** should be of interest as a calculus in its own right. On the practical side, we hoped that System **IF** would allow us to reduce the size of program terms in a type-preserving compiler, and thereby reduce compilation time. In the particular context of the Glasgow Haskell Compiler we successfully demonstrated the former, but not the latter. The balance of costs and benefits might, however, be different in other settings.

As indicated in the related work, there is a rich design space of highly-expressive calculi in which some type information is implicit or abbreviated. Among these, System **IF** appears to be the only one that shares System **F**'s desirable unique-type property. Whether it is possible to elide yet more type information without losing this property remains an open question.

Acknowledgements

We thank Bob Harper, Jeremy Gibbons, Thomas Given-Wilson, Shin-Cheng Mu, Tony Nguyen, Leaf Petersen, Didier Rémy, and the anonymous referees of an earlier submission, for their helpful feedback on drafts of the paper.

Bibliography

Adam Chilpala, Leaf Petersen, and Robert Harper. Strict bidirectional type checking. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'05)*, Long Beach, January 2005. ACM.

Coq. Coq. `http://pauillac.inria.fr/coq/`, 2007.

Neil Ghani. Eta-expansions in dependent type theory – the calculus of constructions. In *Typed Lambda Calculi and Applications*, volume 1210 of *Lecture Notes in Computer Science*, pages 164–180. Springer, 1997.

J-Y Girard. The System F of variable types: fifteen years later. In G Huet, editor, *Logical Foundations of Functional Programming*. Addison-Wesley, 1990.

J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, January 1995.

Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 Haskell Workshop, Montreal*, September 2000. Nottingham University Department of Computer Science Technical Report NOTTCS-TR-00-1.

Barry Jay and Delia Kesner. Pure pattern calculus. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (ed:P. Sestoft)*, pages 100–114, 2006. Revised version at `www-staff.it.uts.edu.au/~cbj/Publications/purepatterns.pdf`.

C.B. Jay. Typing first-class patterns. In *Higher-Order Rewriting, electronic proceedings*, 2006. `http://hor.pps.jussieu.fr/06/proc/jay1.pdf`.

C.B. Jay and N. Ghani. The virtues of eta-expansion. *J. of Functional Programming*, 5(2):135–154, 1995. Also appeared as tech. report ECS-LFCS-92-243.

D Le Botlan and D Rémy. MLF: raising ML to the power of System F. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pages 27–38, Uppsala, Sweden, September 2003. ACM.

R Milner. A theory of type polymorphism in programming. *JCSS*, 13(3), December 1978.

Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 271–283, St Petersburg Beach, Florida, January 1996. ACM.

Tom Murphy. The wizard of TILT: efficent, convenient, and abstract type representations. Technical Report CMU-CS-02-120, Carnegie Mellon University, 2002. URL `http://reports-archive.adm.cs.cmu.edu/anon/2002/CMU-CS-02-120.pdf`.

George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *Logic in Computer Science*, pages 93–104, 1998. URL `citeseer.ist.psu.edu/article/necula98efficient.html`.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, 2007. URL `http://www.cs.chalmers.se/~ulfn/papers/thesis.pdf`.

Leaf Petersen. *Certifying Compilation for Standard ML in a Type Analysis Framework*. PhD thesis, Carnegie Mellon University, 2005. URL `http://reports-archive.adm.cs.cmu.edu/anon/2005/CMU-CS-05-135.pdf`.

Simon Peyton Jones, editor. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

Simon Peyton Jones, Cordelia Hall, Kevin Hammond, Will Partain, and Philip Wadler. The Glasgow Haskell Compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference, Keele*, pages 249–257. DTI/SERC, March 1993. URL {`http://research.microsoft.com/~simonpj/Papers/grasp-jfit.ps.Z`}.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17:1–82, January 2007.

Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 153–163, New York, NY, USA, 1988. ACM. ISBN 0-89791-273-X. doi: http://doi.acm.org/10.1145/62678.62697.

Benjamin C. Pierce and David N. Turner. Local type inference. In *25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 252–265, San Diego, January 1998. ACM.

J. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium '74*, volume 19 of *Lecture Notes in Computer Science*. Springer Verlag, 1974.

B Saha and Z Shao. Optimal type lifting. In *Types in Compilation*, pages 156–177, 1998. URL `citeseer.ist.psu.edu/article/saha98optimal.html`.

Zhong Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.

Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 106–119, Baltimore, 1998. ACM.

D Tarditi, G Morrisett, P Cheng, C Stone, R Harper, and P Lee. TIL: A type-directed optimizing compiler for ML. In *ACM Conference on Programming Languages Design and Implementation (PLDI'96)*, pages 181–192. ACM, Philadelphia, May 1996.

A Tolmach. Tag-free garbage collection using explicit type parameters. In *ACM Symposium on Lisp and Functional Programming*, pages 1–11. ACM, Orlando, Florida, June 1994.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc 16th ACM Symposium on Principles of Programming Languages, Austin, Texas*. ACM, January 1989.

J B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1994.