HMF: Simple Type Inference for First-Class Polymorphism

Daan Leijen
Microsoft Research
daan@microsoft.com

Abstract

HMF is a conservative extension of Hindley-Milner type inference with first-class polymorphism. In contrast to other proposals, HML uses regular System F types and has a simple type inference algorithm that is just a small extension of the usual Damas-Milner algorithm W. Given the relative simplicity and expressive power, we feel that HMF can be an attractive type system in practice. There is a reference implementation of the type system available online together with a technical report containing proofs (Leijen 2007a,b).

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism

General Terms Languages, Design, Theory

Keywords Type Inference, First-class polymorphism

1. Introduction

Type inference in functional languages is usually based on the Hindley-Milner type system (Hindley 1969; Milner 1978; Damas and Milner 1982). Hindley-Milner has a simple logical specification, and a type inference algorithm that can automatically infer most general, or *principal*, types for expressions without any further type annotations.

To achieve automatic type inference, the Hindley-Milner type system restricts polymorphism where function arguments and elements of structures can only be monomorphic. Formally, this means that universal quantifiers can only appear at the outermost level (i.e. higher-ranked types are not allowed), and quantified variables can only be instantiated with monomorphic types (i.e. impredicative instantiation is not allowed). These are severe restrictions in practice. Even though uses of first-class polymorphism occur infrequently, there is usually no good alternative or work around (see (Peyton Jones et al. 2007) for a good overview).

The reference calculus for first-class polymorphism is System F which is explicitly typed. As remarked by Rémy (2005) one would like to have the expressiveness of System F combined with the convenience of Hindley-Milner type inference. Unfortunately, full type inference for System F is undecidable (Wells 1999). Therefore, the only way to achieve our goal is to augment Hindley-Milner type inference with just enough programmer provided annotations to make programming with first-class polymorphism a joyful experience.

There has been quite some research into this area (Peyton Jones et al. 2007; Rémy 2005; Jones 1997; Le Botlan and Rémy 2003; Le Botlan 2004; Odersky and Läufer 1996; Garrigue and Rémy 1999; Vytiniotis et al. 2006; Dijkstra 2005) but no fully satisfactory solution has been found yet. Many proposed systems are quite complex, and use for example algorithmic specifications, or introduce new forms of types that go beyond regular System F types.

In this article, we present HMF, a simple and conservative extension of Hindley-Milner with first-class polymorphism that needs few annotations in practice. The combination of simplicity and expressiveness can make HMF an attractive replacement of Hindley-Milner in practice. In particular:

- HMF is a conservative extension: every program that is well-typed in Hindley-Milner, is also a well-typed HMF program and type annotations are never required for such programs. Through type annotations, HMF supports first-class polymorphic values and impredicative instantiation. Unlike previous works, HMF does not require any new form of types (such of boxed types or flexible bindings) but only uses familiar System F types.
- In practice, few type annotations are needed for programs that go beyond Hindley-Milner. Only polymorphic parameters and ambiguous impredicative instantiations must be annotated. Both cases can be clearly specified and are relatively easy to apply in practice.
- The type inference algorithm is very close to algorithm W (Damas and Milner 1982). It does not require unfamiliar operations, which makes it relatively easy to understand and implement.
- HMF is robust with respect to abstraction. It has the property that whenever the application $e_1\ e_2$ is well-typed, so is the abstraction $apply\ e_1\ e_2$. We consider this an important property as it implies that we can reuse common polymorphic abstractions over general polymorphic values.

In the following section we give an overview of HMF in practice. Section 4 presents the formal logical type rules of HMF followed by a description of the type inference algorithm in Section 6. Finally, Section 5 discusses type annotations in more detail.

2. Overview and background

HMF extends Hindley-Milner with regular System F types where polymorphic values are first-class citizens. To support first-class polymorphism, two ingredients are needed: higher-ranked types and impredicative instantiation.

2.1 Higher-rank types

Hindley-Milner allows definitions to be polymorphic and reused at different type instantiations. Take for example the identity function:

$$id :: \forall\alpha.\,\alpha \to \alpha \quad \text{(inferred)}$$
$$id\ x = x$$

Because this function is polymorphic in its argument type, it can be applied to any value, and the tuple expression $(id\ 1, id\ True)$

where id is applied to both an integer and a boolean value is well-typed. Unfortunately, only definitions can be polymorphic while parameters or elements of structures cannot. We need types of *higher-rank* to allow for polymorphic parameters. Take for example the following program:

$poly\ f = (f\ 1, f\ True)$ (rejected)

This program is rejected in Hindley-Milner since there exists no monomorphic type such that the parameter f can be applied to both an Int and a $Bool$. However, in HMF we can explicitly annotate the parameter with a polymorphic type. For example:

$poly\ (f :: \forall\alpha.\, \alpha \to \alpha) = (f\ 1, f\ True)$

is well-typed in HMF, with type $(\forall\alpha.\, \alpha \to \alpha) \to (Int, Bool)$, and the application $poly\ id$ is well-typed. The inferred type for $poly$ is a higher-rank type since the quantifier is nested inside the function type. Note that the parameter f can be assigned many polymorphic types, for example $\forall\alpha.\, \alpha \to \alpha \to \alpha$, or $\forall\alpha.\, \alpha \to Int$, where neither is an instance of the other. Because of this, HMF can never infer polymorphic types for parameters automatically, and *parameters with a polymorphic type must be annotated*.

Higher-rank polymorphism has many applications in practice, including type-safe encapsulation of state and memory transactions, data structure fusion, and generic programming. For a good overview of such applications we refer the interested reader to (Peyton Jones et al. 2007).

2.2 Impredicative instantiation

Besides higher-rank types, HMF also supports the other ingredient for first-class polymorphism, namely impredicative instantiation, where type variables can be instantiated with polymorphic types (instead of just monomorphic types). We believe that this is a crucial property that enables the use of normal polymorphic abstractions over general polymorphic values. For example, if we define:

$apply :: \forall\alpha\beta.\, (\alpha \to \beta) \to \alpha \to \beta$ (inferred)
$apply\ f\ x = f\ x$

then the expression

$apply\ poly\ id$

is well-typed in HMF, where the type variable α in the type of $apply$ is impredicatively instantiated to the polymorphic type $\forall\alpha.\, \alpha \to \alpha$ (which is not allowed in Hindley Milner). Unfortunately, we cannot always infer impredicative instantiations automatically since this choice is sometimes ambiguous.

Consider the function $single :: \forall\alpha.\, \alpha \to [\alpha]$ that creates a singleton list (where we use the notation $[\alpha]$ for a list of elements of type α). In a predicative system like Hindley-Milner, the expression $single\ id$ has type $\forall\alpha.\, [\alpha \to \alpha]$. In a system with impredicative instantiation, we can also a give it the type $[\forall\alpha.\, \alpha \to \alpha]$ where all elements are kept polymorphic. Unfortunately, neither type is an instance of the other and we have to disambiguate this choice.

Whenever there is an ambiguous impredicative application, HMF always prefers the predicative instantiation, and always introduces the least inner polymorphism possible. Therefore, HMF is by construction fully compatible with Hindley-Milner and the type of $single\ id$ is also $\forall\alpha.\, [\alpha \to \alpha]$ in HMF. If the impredicative instantiation is wanted, a type annotation is needed to make this choice unambiguous. For example, we can create a list of polymorphic identity functions as:[1]

$ids = (single :: (\forall\alpha.\, \alpha \to \alpha) \to [\forall\alpha.\, \alpha \to \alpha])\ id$

[1] We can also write `single (id` $:: \forall\alpha.\, \alpha \to \alpha)$ with rigid type annotations (Section 5.3)

where ids has type $[\forall\alpha.\, \alpha \to \alpha]$. Fortunately, ambiguous impredicative applications only happen in one specific case: namely when a function with a type of the form $\forall\alpha.\, \alpha \to ...$ is applied to a polymorphic argument whose outer quantifiers should not be instantiated (as in $single\ id$). In all other cases, the (impredicative) instantiations are always fully determined and an annotation is never needed. This is the case for example if the function has type $\forall\alpha.\, [\alpha] \to ...$, or if the argument has no outer quantifiers. For example, we can create a singleton list with ids as its element without extra annotations:

$idss :: [[\forall\alpha.\, \alpha \to \alpha]]$ (inferred)
$idss = single\ ids$

since the instantiation is unambiguous. Moreover, HMF considers all arguments in an application to disambiguate instantiations and is not sensitive to the order of arguments. Consider for example reverse application defined as:

$revapp :: \forall\alpha\beta.\, \alpha \to (\alpha \to \beta) \to \beta$ (inferred)
$revapp\ x\ f = f\ x$

The application $revapp\ id\ poly$ is accepted without any annotation as the impredicative instantiation of the quantifier α in the type of $revapp$ to $\forall\alpha.\, \alpha \to \alpha$ is uniquely determined by considering both arguments.

More generally, HMF has the property that whenever an application $e_1\ e_2$ is well typed, than the expression $apply\ e_1\ e_2$ is also well typed, and also the reverse application $revapp\ e_2\ e_1$. We consider this an important property since it applies more generally for arbitrary functors (map) applying polymorphic functions ($poly$) over structures that hold polymorphic values (ids). A concrete example of this that occurs often in practice is the application of $runST$ in Haskell. The function $runST$ executes a state monadic computation in type safe way and its (higher-rank) type is:

$runST :: \forall\alpha.\, (\forall s.\, ST\ s\ \alpha) \to \alpha$

Often, Haskell programmers use the application operator (\$) to apply $runST$ to a large computation as in:

$runST\ \$\ computation$

Given that (\$) has the same type as $apply$, HMF accepts this application without annotation and impredicatively instantiates the α quantifier of $apply$ to $\forall s.\, ST\ s\ \alpha$. In practice, automatic impredicative instantiation ensures that we can also reuse many common abstractions on structures with polymorphic values without extra annotations. For example, we can apply $length$ to a list with polymorphic elements,

$length\ ids$

or map the $head$ function over a list of lists with polymorphic elements,

$map\ head\ (single\ ids)$

or similarly:

$apply\ (map\ head)\ (single\ ids)$

without giving any type annotation.

2.3 Robustness

HMF is not entirely robust against small program transformations and sometimes requires the introduction of more annotations. In particular, η-expansion does not work for polymorphic parameters since these must always be annotated in HMF. For example, $\lambda f.poly\ f$ is rejected and we should write instead $\lambda(f :: \forall\alpha.\, \alpha \to \alpha).poly\ f$.

Moreover, since HMF disambiguates impredicative instantiations over multiple arguments at once, we cannot always abstract over partial applications without giving an extra annotation. For example, even though $revapp\ \ id\ \ poly$ is accepted, the 'equivalent' program $\textbf{let}\ f = revapp\ \ id\ \textbf{in}\ f\ \ poly$ is not accepted without an extra annotation, since the type assigned to the partial application $revapp\ \ id$ in isolation is the Hindley-Milner type $\forall\alpha\beta.\,((\alpha \to \alpha) \to \beta) \to \beta$ and the body $f\ \ poly$ is now rejected.

Nevertheless, we consider the latter program as being quite different from a type inference perspective since the partial application $revapp\ \ id$ can now be potentially shared through f with different (polymorphic) types. Consider for example $\textbf{let}\ f = revapp\ \ id\ \textbf{in}\ (f\ \ poly, f\ \ iapp)$ where $iapp$ has type $(Int \to Int) \to Int \to Int$. In this case, there does not exist any System F type for f to make this well-typed, and as a consequence we must reject it. HMF is designed to be modular and to stay firmly within regular System F types. Therefore f gets assigned the regular Hindley-Milner type. If the polymorphic instantiation is wanted, an explicit type annotation must be given.

3. A comparision with MLF and boxy types

In this section we compare HMF with two other type inference systems that support first-class polymorphism, namely MLF (Le Botlan and Rémy 2003; Le Botlan 2004; Le Botlan and Rémy 2007; Rémy and Yakobowski 2007) and boxy type inference (Vytiniotis et al. 2006).

MLF

The MLF type system also supports full first-class polymorphism, and only requires type annotations for parameters that are used polymorphically. As a consequence, MLF is strictly more powerful than HMF, and every well-typed HMF program is also a well-typed MLF program. MLF achieves this remarkable feat by going beyond regular System F types and introduces polymorphically bounded types. This allows MLF to 'delay' instantiation and give a principal type to ambiguous impredicative applications. For example, in the program $\textbf{let}\ f = revapp\ \ id\ \textbf{in}\ (f\ \ poly, f\ \ iapp)$, the type assigned to f is $\forall(\gamma \geqslant \forall\alpha.\,\alpha \to \alpha).\,\forall\beta.\,(\gamma \to \beta) \to \beta$, which can be instantiated to either $\forall\beta.\,((\forall\alpha.\,\alpha \to \alpha) \to \beta) \to \beta$ or $\forall\alpha\beta.\,((\alpha \to \alpha) \to \beta) \to \beta$. Since applications never need an annotation, this makes MLF robust under rewrites. For example, when the application $e_1\ \ e_2$ is well-typed, than so is $apply\ \ e_1\ \ e_2$ and also $revapp\ \ e_2\ \ e_1$, and partial applications can always be abstracted by a let-binding.

As shown in Section 2.1, inference for polymorphic parameters is not possible in general and we can therefore argue that MLF is an upper bound in the design space which achieves optimal (local) type inference in the sense that it requires the minimal number of annotations possible. The drawback of MLF is that it goes beyond regular System F types which makes MLF considerably more complicated. This is not only the case for programmers that have to understand these types, but also for the meta theory of MLF, the implementation of the type inference algorithm, and the translation to System F (which is important for qualified types (Leijen 2007c; Leijen and Löh 2005)).

HMF represents a lower bound in the design space and only uses regular System F types. As shown in Section 2.2, HMF does this at the price of also requiring annotations on ambiguous impredicative applications which is harder on the programmer. In return for those annotations though, we get a simpler system with familiar System F types and where the inference algorithm is a small extension of algorithm W (which also makes it easier to extend HMF with qualified types for example).

Boxy type inference

The GHC compiler supports first-class polymorphism using boxy type inference. This inference system is made principal by distinguishing between inferred 'boxy types' and checked annotated types. There are actually two variants of boxy type inference, namely basic boxy type inference, and the extension with 'pre-subsumption' (Vytiniotis et al. 2006, Section 6). The basic version is quite weak cannot type simple applications like $tail\ ids$ or propagate the annotation in $single\ \ id :: [\forall\alpha.\,\alpha \to \alpha]$. Therefore, we only discuss the extended version with pre-subsumption (which is implemented in GHC).

Unfortunately, there are no clear rules for programmers when annotations are needed with boxy type inference. In general, it is hard to characterize those situations precisely since they depend on the typing context, and the details of the boxy matching and pre-subsumption algorithms.

In general, most polymorphic parameters and impredicative applications need an annotation with boxy type inference. However, due to the built-in type propagation, we can often just annotate the result type, as in $(single\ \ id) :: [\forall\alpha.\,\alpha \to \alpha]$ (which is rejected in HMF). Annotations can also be left out when the type is apparent from the context, as in $foo\ \ (\lambda f.(f\ \ 1, f\ \ True))$ where foo has type $((\forall\alpha.\,\alpha \to \alpha) \to (Int, Bool)) \to Int$. Neither HMF nor MLF can type this example and need an annotation on f. Of course, local propagation of types is not robust under small program transformations. For example, the abstraction $\textbf{let}\ poly = \lambda f.(f\ \ 1, f\ \ True)\ \textbf{in}\ foo\ \ poly$ is not well-typed and the parameter f needs to be annotated in this case.

In contrast to HMF, annotations are sometimes needed even if the applications are unambiguous. Take for example the function $choose$ with type $\forall\alpha.\,\alpha \to \alpha \to \alpha$, and the empty list $null$ with type $\forall\alpha.\,[\alpha]$. Both the applications $choose\ \ null\ \ ids$ and $choose\ ids\ null$ are rejected with boxy type inference even though the instantiations are unambiguous[2]. Surprisingly, the abstraction $\textbf{let}\ f = choose\ \ null\ \textbf{in}\ f\ \ ids$ is accepted due to an extra generalization step on let bindings. All of these examples are accepted without annotations in both HMF and MLF.

Finally, even if an impredicative application $e_1\ \ e_2$ is accepted, the abstraction $apply\ \ e_1\ \ e_2$ (and $revapp\ \ e_2\ e_1$) is still rejected with boxy type inference without an extra type annotation. For example, the application $apply\ \ runST\ \ (return\ \ 1)$ must be annotated as $(apply :: ((\forall s.\,ST\ \ s\ \ Int) \to Int) \to (\forall s.\,ST\ \ s\ \ Int) \to Int)\ \ runST\ \ (return\ 1)$. We feel that this can be a heavy burden in general when abstracting over common polymorphic patterns.

$$
\begin{array}{lll}
\sigma ::= \forall\alpha.\,\sigma & & \text{(quantified type)} \\
\quad\ \mid \alpha & & \text{(type variable)} \\
\quad\ \mid c\ \sigma_1 \ldots \sigma_n & & \text{(type constructor application)} \\
 & & \\
\rho ::= \alpha \mid c\ \sigma_1 \ldots \sigma_n & & \text{(unquantified types)} \\
\tau ::= \alpha \mid c\ \tau_1 \ldots \tau_n & & \text{(monomorphic types)}
\end{array}
$$

Figure 1. HMF types

4. Type rules

HMF uses regular System F types as defined Figure 1. A type σ is either a quantified type $\forall\alpha.\,\sigma$, a type variable α, or the application of a type constructor c. Since HMF is invariant, we do not treat the function constructor $(\to)$ specially and assume it is part of the type constructors c. The free type variables of a type σ are denoted as $ftv(\sigma)$:

[2] GHC actually accepts the second expression due to a left-to-right bias in type propagation.

$$\begin{array}{ll} ftv(\alpha) & = \{\alpha\} \\ ftv(c\ \sigma_1 \ldots \sigma_n) & = ftv(\sigma_1) \cup \ldots \cup ftv(\sigma_n) \\ ftv(\forall\alpha.\,\sigma) & = ftv(\sigma) - \{\alpha\} \end{array}$$

and is naturally extended to larger constructs containing types.

In the type rules, we sometimes distinguish between polymorphic types σ and monomorphic types. Figure 1 defines unquantified types ρ as types without an outer quantifier, and monomorphic types τ as types without any quantifiers at all (which correspond to the usual Hindley-Milner τ types).

4.1 Substitution

A substitution S is a function that maps type variables to types. The empty substitution is the identity function and written as $[\,]$. We write Sx for the application of a substitution S to x where only the free type variables in x are substituted. We often write a substitution as a finite map $[\alpha_1 := \sigma_1, ..., \alpha_n := \sigma_n]$ (also written as $[\overline{\alpha} := \overline{\sigma}]$) which maps α_i to σ_i and all other type variables to themselves. The domain of a substitution contains all type variables that map to a different type: $dom(S) = \{\alpha \mid S\alpha \neq \alpha\}$. The codomain is a set of types and defined as: $codom(S) = \{S\alpha \mid \alpha \in dom(S)\}$. We write $(\alpha := \sigma) \in S$ if $\alpha \in dom(S)$ and $S\alpha = \sigma$. The expression $(S - \overline{\alpha})$ removes $\overline{\alpha}$ from the domain of S, i.e. $(S - \overline{\alpha}) = [\alpha := \sigma \mid (\alpha := \sigma) \in S \wedge \alpha \notin \overline{\alpha}]$. Finally, we only consider *idempotent* substitutions S where $S(Sx) = Sx$ (and therefore $ftv(codom(S)) \not\cap dom(S)$).

4.2 Type instance

We use the regular System F polymorphic *generic instance* relation ($\sqsubseteq$) on types, defined as:

$$\frac{\overline{\beta} \not\cap ftv(\forall\overline{\alpha}.\,\sigma_1)}{\forall\overline{\alpha}.\,\sigma_1 \sqsubseteq \forall\overline{\beta}.\,[\overline{\alpha} := \overline{\sigma}]\sigma_1}$$

where we write ($\not\cap$) for disjoint sets. Note that the generic instance relation can only instantiate the outer *bound* variables. Here are some examples:

$$\begin{array}{lcl} \forall\alpha.\,\alpha\to\alpha & \sqsubseteq & Int \to Int \\ \forall\alpha.\,\alpha\to\alpha & \sqsubseteq & \forall\beta.\,[\forall\alpha.\,\alpha\to\beta] \to [\forall\alpha.\,\alpha\to\beta] \end{array}$$

Note that HMF is invariant since the instance relation can only instantiate outer quantifiers. Two types are considered equal if they are instances of each other:

$$\sigma_1 = \sigma_2 \ \triangleq\ (\sigma_1 \sqsubseteq \sigma_2 \wedge \sigma_2 \sqsubseteq \sigma_1)$$

This means that we can freely apply α-renaming, reorder quantifiers, and that unbound quantifiers are irrelevant. Finally, we write $[\![\sigma]\!]$ for the *polymorphic weight* of a type, which is defined as the number of inner quantifiers:

$$\begin{array}{l} [\![\forall\overline{\alpha}.\,\rho]\!] = wt(\rho) \\ \quad \text{where} \\ \qquad \begin{array}{lll} wt(\alpha) & = 0 & \\ wt(c\ \sigma_1 \ldots \sigma_n) & = wt(\sigma_1) + \ldots + wt(\sigma_n) + 0 & \\ wt(\forall\alpha.\,\sigma) & = wt(\sigma) & \text{iff}\quad \alpha \notin ftv(\sigma) \\ wt(\forall\alpha.\,\sigma) & = wt(\sigma) + 1 & \text{otherwise} \end{array} \end{array}$$

and extends naturally to structures containing types. For example, $[\![\forall\alpha.\,[\forall\beta.\,\alpha \to \beta]]\!]$ is one, while $[\![\tau]\!]$, the polymorphic weight of monomorphic types, is always zero. Note that the polymorphic weight is monotonically increasing with respect to instantiation, i.e.

Property 1 (*Polymorphic weight is stable*):

If $\sigma_1 \sqsubseteq \sigma_2$ then $[\![\sigma_1]\!] \leqslant [\![\sigma_2]\!]$

The polymorphic weight is used in the type rules to restrict derivations to have a minimal polymorphic weight, effectively preventing the introduction of arbitrary polymorphic types.

$$\begin{array}{ll}
\textsc{Var} & \dfrac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \\[2ex]
\textsc{Gen} & \dfrac{\Gamma \vdash e : \sigma \quad \alpha \notin ftv(\Gamma)}{\Gamma \vdash e : \forall\alpha.\,\sigma} \\[2ex]
\textsc{Inst} & \dfrac{\Gamma \vdash e : \sigma_1 \quad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash e : \sigma_2} \\[2ex]
\textsc{Fun} & \dfrac{\Gamma, x : \tau \vdash e : \rho}{\Gamma \vdash \lambda x.e : \tau \to \rho} \\[2ex]
\textsc{Fun-Ann} & \dfrac{\Gamma, x : \sigma \vdash e : \rho}{\Gamma \vdash \lambda(x :: \sigma).e : \sigma \to \rho} \\[2ex]
\textsc{Let} & \dfrac{\begin{array}{c}\Gamma \vdash e_1 : \sigma_1 \quad \Gamma, x : \sigma_1 \vdash e_2 : \sigma_2 \\ \forall\sigma_1'.\ \Gamma \vdash e_1 : \sigma_1' \ \Rightarrow\ \sigma_1 \sqsubseteq \sigma_1'\end{array}}{\Gamma \vdash\ \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \sigma_2} \\[2ex]
\textsc{App} & \dfrac{\begin{array}{c}\Gamma \vdash e_1 : \sigma_2 \to \sigma \quad \Gamma \vdash e_2 : \sigma_2 \\ (\forall\sigma'\sigma_2'.\,(\Gamma \vdash e_1 : \sigma_2' \to \sigma' \wedge \Gamma \vdash e_2 : \sigma_2') \\ \Rightarrow\ [\![\sigma_2 \to \sigma]\!] \leqslant [\![\sigma_2' \to \sigma']\!])\end{array}}{\Gamma \vdash e_1\ e_2 : \sigma}
\end{array}$$

Figure 2. Type rules for Plain HMF

4.3 Type rules

We first describe a simpler version of HMF, called Plain HMF, that does not consider multiple argument applications. In Section 4.5 we describe the addition of a type rule for N-ary applications that is used for full HMF.

The type rules for Plain HMF are given in Figure 2. The expression $\Gamma \vdash e : \sigma$ implies that under a type environment Γ we can assign a type σ to the expression e. The type environment Γ binds variables to types, where we use the expression $\Gamma, x : \sigma$ to extend the environment Γ with a new binding x with type σ (replacing any previous binding for x). Expressions e in HMF are standard and consist of variables x, applications $e_1\ \ e_2$, functions $\lambda x.e$, functions with an annotated parameter $\lambda(x :: \sigma).e$, and local bindings $\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$.

An important property for HMF is the existance of principal type derivations, i.e. for any derivation $\Gamma \vdash e : \sigma'$, there also exists a derivation $\Gamma \vdash e : \sigma$ with a unique most general type σ such that $\sigma \sqsubseteq \sigma'$. In Section 6 we describe a type inference algorithm that infers precisely those principal types and is sound and complete with respect to the type rules.

The rules Var and Gen are standard and equivalent to the usual Hindley-Milner rules. The instantiation rule Inst is generalized to use the System F generic instance relation.

Just like Hindley-Milner, the function rule Fun restricts the type of the parameter x to a monomorphic type τ. As we have seen in the introduction, this is essential to avoid guessing polymorphic types for parameters. Furthermore, the type of the function body must be an unquantified type ρ. For example the expression $\lambda x.\lambda y.x$ has the principal type $\forall\alpha\beta.\,\alpha \to \beta \to \alpha$ in HMF. Without the restriction to unquantified types, the type $\forall\alpha.\,\alpha \to (\forall\beta.\,\beta \to \alpha)$ could also be derived for this expression, and since neither of these types is an instance of each other, we would no longer have principal type derivations.

In contrast, rule Fun-Ann binds the type of the parameter to a given polymorphic type σ. Again, the type of the function body must be an unquantified type ρ. For simplicity we consider only closed annotations in Plain HMF but we remove this restriction in

Section 5.1. There is no special rule for type annotations since we can treat a type annotation $(e :: \sigma)$ as an application to an annotated identity function: $(\lambda(x :: \sigma).x)\ \ e$. Using this encoding, we can derive the following rule for closed annotations:

$$\text{ANN}^{\star}\quad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash (e :: \sigma) : \sigma}$$

using INST, GEN, FUN-ANN, and APP.

The LET rule and application rule APP are standard except for their extra side conditions. Without these conditions the type rules are still sound and would reside between HMF and implicitly typed System F. Unfortunately this system would not have principal type derivations which precludes efficient type inference. The side conditions are therefore pragmatically chosen to be the simplest conditions such that HMF has principal type derivations, simple rules for type annotations, and a straightforward type inference algorithm.

The application rule APP requires that the argument and parameter type are syntactically equivalent which can be full polymorphic types. Furthermore, the rule requires that the polymorphic weight of the function type is minimal, i.e. for any derivations $\Gamma \vdash e_1 : \sigma_2' \to \sigma'$ and $\Gamma \vdash e_2 : \sigma_2'$, we have that $[\![\sigma_2 \to \sigma]\!] \leqslant [\![\sigma_2' \to \sigma']\!]$. For convenience, we often use the shorthand $\mathit{minimal}([\![\sigma_2 \to \sigma]\!])$ to express this condition. Note that for monomorphic applications, the polymorphic weight is always zero and therefore always minimal. Effectively, the condition ensures that predicative instantation is preferred when possible and that no arbitrary polymorphism can be introduced. Take for example the derivation of the application *single id* from the introduction:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\Gamma \vdash \mathit{single} : \forall\alpha.\,\alpha \to [\alpha]\\ \forall\alpha.\,\alpha \to [\alpha] \sqsubseteq (\alpha\to\alpha)\to[\alpha\to\alpha]\end{array}}{\Gamma \vdash \mathit{single} : (\alpha\to\alpha)\to[\alpha\to\alpha]} \quad \dfrac{\begin{array}{c}\Gamma \vdash \mathit{id} : \forall\alpha.\,\alpha\to\alpha\\ \forall\alpha.\,\alpha\to\alpha \sqsubseteq \alpha\to\alpha\end{array}}{\Gamma \vdash \mathit{id} : \alpha\to\alpha} \quad \mathit{minimal}([\![(\alpha\to\alpha)\to[\alpha\to\alpha]]\!])}{\Gamma \vdash \mathit{single}\ \mathit{id} : [\alpha\to\alpha] \qquad \alpha \notin \mathit{ftv}(\Gamma)}}{\Gamma \vdash \mathit{single}\ \mathit{id} : \forall\alpha.\,[\alpha\to\alpha]}$$

Without the condition for minimal polymorphic weights, the type $[\forall\alpha.\,\alpha \to \alpha]$ could also be derived for the application $\mathit{single}\ \mathit{id}$:

$$\dfrac{\dfrac{\begin{array}{c}\Gamma \vdash \mathit{single} : \forall\alpha.\,\alpha\to[\alpha]\\ \forall\alpha.\,\alpha\to[\alpha] \sqsubseteq (\forall\alpha.\,\alpha\to\alpha)\to[\forall\alpha.\,\alpha\to\alpha]\end{array}}{\begin{array}{c}\Gamma \vdash \mathit{single} : (\forall\alpha.\,\alpha\to\alpha)\to[\forall\alpha.\,\alpha\to\alpha]\\ \Gamma \vdash \mathit{id} : \forall\alpha.\,\alpha\to\alpha\end{array}}}{\Gamma \vdash \mathit{single}\ \mathit{id} : [\forall\alpha.\,\alpha\to\alpha]} \qquad \text{wrong!}$$

where we would lose principal type derivations since the types $\forall\alpha.\,[\alpha \to \alpha]$ and $[\forall\alpha.\,\alpha \to \alpha]$ are not in an instance relation. The minimality condition ensures that the second derivation is disallowed, since the polymorphic weight $[\![\forall\alpha.\,[\alpha\to\alpha]]\!]$ is smaller than $[\![[\forall\alpha.\,\alpha\to\alpha]]\!]$.

It is important that the minimality condition ranges over the entire sub derivations of e_1 and e_2 since the 'guessed' polymorphism of the second derivation is introduced higher up the tree in the instantiation rule. As shown in these derivations, the condition disambiguates precisely those impredicative applications where a function of type $\alpha \to ...$ is applied to a polymorphic argument. It is easy to see that the argument is always be (predicatively) instantiated in this case (if no annotation was given).

Just like Hindley-Milner, the LET rule derives a polymorphic type for let-bound values. In addition, the rule requires that the type of the bound value is the most general type that can be derived, i.e. for any derivation $\Gamma \vdash e_1 : \sigma_1'$, we have that $\sigma_1 \sqsubseteq \sigma_1'$. As a convenient shorthand, we often write $\mathit{mostgen}(\sigma_1)$ for this condition.

The condition on let bindings is required to prevent the introduction of arbitrary polymorphism through polymorphic types in the type environment Γ. Without it, we could for example bind single' to *single* with the (polymorphically) instantiated type $(\forall\alpha.\,\alpha \to \alpha) \to [\forall\alpha.\,\alpha \to \alpha]$, and derive for the application $\mathit{single}'\ \mathit{id}$ the type $[\forall\alpha.\,\alpha \to \alpha]$ and lose principal type derivations again.

We cannot just require that the let-bound values are of minimal polymorphic weight as in the application rule, since arbitrary polymorphism can also be introduced through the sharing of quantified type variables. Consider the expression ($\mathbf{let}\ \mathit{foo}\ x\ y = \mathit{single}\ y\ \mathbf{in}\ \mathit{foo}\ \mathit{ids}\ \mathit{id}$) where *ids* has type $[\forall\alpha.\,\alpha \to \alpha]$. The principal type for this expression is $\forall\alpha.\,[\alpha \to \alpha]$, where the type for *foo* is $\forall\alpha\beta.\,\beta \to \alpha \to [\alpha]$. Without the most general type restriction, we could also assign the type $\forall\alpha.\,[\alpha] \to \alpha \to [\alpha]$ to *foo* and through arbitrary sharing derive the incomparable type $[\forall\alpha.\,\alpha \to \alpha]$ for the expression.

The type rules of HMF allow principal derivations and are sound where well-typed programs cannot go 'wrong'. We can prove this by showing that for every HMF derivation there is a corresponding System F term that is well-typed (Leijen 2007b). Furthermore, HMF is a conservative extension of Hindley-Milner. In Hindley-Milner programs rule FUN-ANN does not occur and all instantiations are monomorphic. This implies that the types in an application are always monomorphic and therefore the minimality restriction is always satisfied. Since Hindley-Milner programs have principal types, we can also always satisfy the most general types restriction on let bindings. Finally, it is interesting that if we just restrict instantiation to monomorphic instantiation, we end up with a predicative type system for arbitrary rank type inference (Peyton Jones et al. 2007; Odersky and Läufer 1996).

4.4 On the side conditions

The LET rule restriction to most-general types is not new. It has been used for example in the typing of dynamics in ML (Leroy and Mauny 1991), local type inference for $F_{\leqslant}$ (Pierce and Turner 1998), semi-explicit first-class polymorphism (Garrigue and Rémy 1999), and more recently for boxy type inference (Vytiniotis et al. 2006). All of these systems require some form of minimal solutions in order to have principal type derivations.

From a logical perspective though, the conditions on LET and APP are unsatisfactory since they range over all possible derivations at that point and can therefore be more difficult to reason about (even though they are still inductive). There exists a straighforward decision procedure however to fullfill the conditions by always using most general type derivations. This automatically satisfies the LET rule side condition, and due to Property 1 will also satisfy the minimality condition on the APP rule where only rule INST on e_1 and e_2 needs to be considered (which is a key property to enable efficient type inference).

It is interesting to note that the type rules without the side conditions are still sound, but would lack principal derivations, and the type inference algorithm would be incomplete. This is the approach taken by Pierce and Turner (1998) for local type inference for example which is only partially complete.

Even though we are not fully satisfied with the side conditions from a logical perspective, we believe that the specification is still natural from a programmers perspective, with clear rules when annotations are needed. Together with the use of just regular System F types and a straightforward type inference algorithm, we feel that the practical advantages justify the use of these conditions in the specification of the type rules.

4.5 N-ary applications

Since Plain HMF requires minimal polymorphic weight on every application node, it is sensitive to the order of the applications. For example, if $e_1\ \ e_2$ is well-typed, so is $\mathit{apply}\ \ e_1\ \ e_2$, but the reverse application, $\mathit{revapp}\ \ e_2\ \ e_1$ is not always accepted. As a concrete example, $revapp\ id\ poly$ is rejected since the principal type of the application $revapp\ id$ in Plain HMF is $\forall\alpha\beta.\,(\alpha \to \alpha) \to \beta \to \beta$ and we cannot derive the (desired) type $\forall\beta.\,(\forall\alpha.\,\alpha \to \alpha) \to \beta \to \beta$ since its polymorphic weight is larger.

A solution to this problem is to allow the application rule to have a minimal polymorphic weight over multiple arguments. In particular, we extend Plain HMF to full HMF by adding the following rule for N-ary applications:

App-N

$$\frac{\begin{array}{c}\Gamma \vdash e : \sigma_1 \to \ldots \to \sigma_n \to \sigma \quad \Gamma \vdash e_1 : \sigma_1 \quad \ldots \quad \Gamma \vdash e_n : \sigma_n \\ \forall \sigma' \sigma'_1..\sigma'_n.\ \ \Gamma \vdash e : \overrightarrow{\sigma_n'} \to \sigma' \wedge \Gamma \vdash e_1 : \sigma'_1 \wedge .. \wedge \Gamma \vdash e_n : \sigma'_n \\ \Rightarrow\ [\![\overrightarrow{\sigma_n} \to \sigma]\!] \leqslant [\![\overrightarrow{\sigma_n'} \to \sigma']\!]\end{array}}{\Gamma \vdash e\ e_1 \ldots e_n : \sigma}$$

where we write $\overrightarrow{\sigma_n}$ for the type $\sigma_1 \to \ldots \to \sigma_n$. With the rule App-N, it becomes possible to accept the application $revapp\ id\ poly$ since we can instantiate $revapp$ to $(\forall\alpha.\,\alpha \to \alpha) \to ((\forall\alpha.\,\alpha \to \alpha) \to (Int, Bool)) \to (Int, Bool)$ which has a minimal polymorphic weight when both arguments are considered.

Even though it is always best to consider the maximal number of arguments possible, the rule App-N does not require to always consider all arguments in an application, and derivations for partial applications are still possible. In fact, it would be wrong to always consider full applications since functions can return polymorphic functions that need to be instantiated first using rule Inst. As an example, consider the expression $head\ ids\ 1$. For this application, it is essential to consider the application $head\ ids$ first in order to use Inst to instantiate its polymorphic result $\forall\alpha.\,\alpha \to \alpha$ to the required $Int \to Int$ type, and we cannot use App-N directly.

5. About type annotations

In principle HMF does not need any special rules for type annotations since we can type an annotation $(e :: \sigma)$ as an application to a typed identity function: $(\lambda(x :: \sigma).x)\ e$. However, in practice it is important to handle annotations with free variables and to propagate type annotation information to reduce the annotation burden. In this section we discuss these issues in more detail. Note that all three techniques described in this section are not required for HMF as such but make it more convenient to work with in practice. All of these concepts can be applied in general to any Hindley-Milner based type inference systems.

5.1 Partial annotations

In order to give types to any subexpression, we need to be able to give *partial type annotations* (Rémy 2005). We write $e :: \exists\overline{\alpha}.\,\sigma$ for a partial type annotation where the free variables $\overline{\alpha}$ in σ are locally bound. We read the annotation as “for some (monomorphic) types $\overline{\alpha}$, the expression e has type σ” (and therefore call $\exists$ the ‘some’ quantifier). As a practical example of such annotation, consider the type of $runST$:

$$runST :: \forall\alpha.\,(\forall s.\,ST\ s\ \alpha) \to \alpha$$

If we define this function, the parameter needs a partial annotation:

$$runST\ (x :: \exists\alpha.\forall s.\,ST\ s\ \alpha) = \ldots$$

Note that we cannot annotate the parameter as $\forall\alpha s.\,ST\ s\ \alpha$ since the parameter itself is not polymorphic in α. For simplicity, we still require type annotations to be closed but of course it is possible to extend this with *scoped type variables* (Peyton Jones and Shields 2004), where annotations can contain free type variables that are bound elsewhere.

We can formalize partial annotations in the type rules by modifying the annotation rule to assume fresh monotypes for the ‘some’ quantifiers:

$$\textsc{Fun-Ann}\ \ \frac{\sigma_2 = [\overline{\alpha} := \overline{\tau}]\sigma_1 \quad \Gamma, x : \sigma_2 \vdash e : \rho}{\Gamma \vdash \lambda(x :: \exists\overline{\alpha}.\,\sigma_1).e : \sigma_2 \to \rho}$$

Moreover, we can remove the Fun rule since we can encode unannoted functions $\lambda x.e$ as $\lambda(x :: \exists\alpha.\,\alpha).e$. Using this encoding, Gen, and Fun-Ann, we can derive the following rule for unannoted functions:

$$\textsc{Fun}^{\star}\ \ \frac{\Gamma \vdash \lambda(x :: \tau).e : \sigma}{\Gamma \vdash \lambda x.e : \sigma}$$

5.2 Type annotation propagation

$$\begin{array}{l}(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) :: \exists\overline{\alpha}.\,\sigma \\ \quad \rightsquigarrow (\mathbf{let}\ x = e_1\ \mathbf{in}\ (e_2 :: \exists\overline{\alpha}.\,\sigma)) :: \exists\overline{\alpha}.\,\sigma \\ (\lambda x.e) :: \exists\overline{\alpha}.\forall\overline{\beta}.\,\sigma_1 \to \sigma_2 \\ \quad \rightsquigarrow (\lambda(x :: \exists\overline{\alpha}\overline{\beta}.\,\sigma_1).(e :: \exists\overline{\alpha}\overline{\beta}.\,\sigma_2)) :: \exists\overline{\alpha}.\forall\overline{\beta}.\,\sigma_1 \to \sigma_2\end{array}$$

Figure 3. Type annotation propagation

Another important addition in practice is the propagation of type annotations. For example, a programmer might write the following definition for $poly$:

$$\begin{array}{l} poly :: (\forall\alpha.\,\alpha \to \alpha) \to (Int, Bool) \\ poly\ f = (f\ 1, f\ True)\end{array}$$

As it stands, this would be rejected by HMF since the parameter f itself is not annotated (and used polymorphically). We can remedy this situation by propagating the type annotation down through lambda and **let** expressions. Figure 3 defines rules for propagating type information, where a rule $e_1 :: \sigma \rightsquigarrow e_2 :: \sigma$ propagates the type annotation on e_1 into a newly annotated expression e_2. The specified rules should be applied recursively to all parts of an expression until no further progress is possible. Note that the rules leave all original annotations in place. Also, the propagation is conservative and the propagated types can be less precise. For example, in the propagation for lambda expressions the sharing of the type variables $\overline{\beta}$ is not propagated. As a practical example, the above expression would be transformed into:

$$\begin{array}{l} poly :: (\forall\alpha.\,\alpha \to \alpha) \to (Int, Bool) \\ poly\ (f :: \forall\alpha.\,\alpha \to \alpha) = (f\ 1, f\ True) :: (Int, Bool)\end{array}$$

and the definition is now well-typed in HMF. Type propagation can be seen as preprocessing step since it is defined as a separate syntactical transformation, and can be understood separately from the order independent specification of the type rules. We consider this an important property since systems that combine type propagation with type inference lead to algorithmic formulations of the type rules that are fragile and difficult to reason about (Rémy 2005).

5.3 Rigid annotations

In general, we cannot statically propagate types through application nodes (since the expression type can be more polymorphic than the propagated type). This is a serious weakness in practice. Consider again the definition of ids from the introduction:

$$(single :: (\forall\alpha.\,\alpha \to \alpha) \to ([\forall\alpha.\,\alpha \to \alpha]))\ id$$

In a system that mixes type propagation with type inference, like boxy type inference (Vytiniotis et al. 2006), we could write instead:

$$(single\ id) :: [\forall\alpha.\,\alpha \to \alpha] \quad \text{(rejected in HMF)}$$

Even though this looks natural and can be implemented for HMF too, we will not give in to the siren call of mixing type propagation with type inference and stick with a declarative formulation of the type rules. Instead, we propose to make type annotations *rigid*. In particular, when a programmer writes a type annotation on an argument or the body of a lambda expression, we will take the type literally and not instantiate or generalize it further. This mechanism allows the programmer to write an annotation on an argument instead of a function, and we can write:

$$\mathit{single}\ (\mathit{id} :: \forall\alpha.\, \alpha \to \alpha)$$

which has type $[\forall\alpha.\, \alpha \to \alpha]$. We believe that rigid annotations are a good compromise to avoid an algorithmic specification of the type system. Moreover, we appreciate the ability to be very specific about the type of an expression where rigid annotations give precise control over type instantiation. For example, we can write a variation of the *const* function that returns a polymorphic function:

$$\begin{array}{l} \mathit{const}' :: \forall\alpha.\, \alpha \to (\forall\beta.\, \beta \to \alpha) \quad \text{(inferred)} \\ \mathit{const}'\ x = (\lambda y \to x) :: \exists\alpha.\, \forall\beta.\, \beta \to \alpha \end{array}$$

Note that with the type annotation propagation of Figure 3 we can also write:

$$\begin{array}{l} \mathit{const}' :: \forall\alpha.\, \alpha \to (\forall\beta.\, \beta \to \alpha) \\ \mathit{const}'\ x\ y = x \end{array}$$

Note that rigid annotations are generally useful and are not specific to HMF and we believe that expression annotations in any language based on Hindley-Milner should be treated rigidly.

Rigid annotations can be formalized with ease using simple syntactic restrictions on the derivations. First we consider an expression to be annotated when it either has a direct annotation or if it is a let expression with an annotated body. The grammar for annotated expressions e_a is:

$$e_a ::= e :: \sigma \mid \mathbf{let}\ x = e\ \mathbf{in}\ e_a$$

Dually, we define unannotated expressions e_u as all other expressions, namely:

$$e_u ::= x \mid e_1\ e_2 \mid \lambda x.e \mid \lambda(x :: \sigma).e \mid \mathbf{let}\ x = e\ \mathbf{in}\ e_u$$

We want to treat annotated expressions rigidly and not instantiate or generalize their types any further. Therefore, our first adaptation to the type rules of Figure 2 is to restrict instantiation and generalization to unannotated expressions only:

$$\textsc{Inst}\ \frac{\Gamma \vdash e_u : \sigma_1 \quad \sigma_1 \sqsubseteq \sigma_2}{\Gamma \vdash e_u : \sigma_2} \qquad \textsc{Gen}\ \frac{\Gamma \vdash e_u : \sigma \quad \alpha \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash e_u : \forall\alpha.\, \sigma}$$

Since instantiation and generalization are now restricted to unannotated expressions, we can instantly derive the type $[\forall\alpha.\, \alpha \to \alpha]$ for the application $\mathit{single}\ (\mathit{id} :: \forall\alpha.\, \alpha \to \alpha)$ since the minimal weight condition of rule App is now satisfied. At the same time, the application $(\mathit{id} :: \forall\alpha.\, \alpha \to \alpha)\ 42$ is now rejected – indeed, a correct annotation would rather be $(\mathit{id} :: \exists\alpha.\, \alpha \to \alpha)\ 42$.

Moreover, we can allow lambda bodies to have a polymorphic type as long as the body expression is annotated, and we add an extra rule for lambda expressions with annotated bodies:

$$\textsc{Fun-Ann-Rigid}\ \frac{\Gamma, x : \sigma_1 \vdash e_a : \sigma_2}{\Gamma \vdash \lambda(x :: \sigma_1).e_a : \sigma_1 \to \sigma_2}$$

Note that we don't need such rule for unannoted functions as $\textsc{Fun}^{\star}$ can be used with both Fun-Ann and Fun-Ann-Rigid.

5.4 Translation of System F to HMF

Plain HMF can almost express any System F program, except for terms that return polymorphic values from a function since Plain HMF restricts function results to ρ types only. We do not see this as a serious restriction as such System F terms are always $\beta\eta$-convertible to a term with a prenex type, i.e. such terms do not add any significant expressive power.

$$\begin{array}{lll}
\mathcal{F}[\![\mathsf{x}]\!]_\Gamma & = x & \\
\mathcal{F}[\![\Lambda\alpha.\, \mathsf{e}]\!]_\Gamma & = \mathcal{F}[\![\mathsf{e}]\!]_\Gamma & \\
\mathcal{F}[\![\mathsf{e}\ \sigma]\!]_\Gamma & = \mathcal{F}[\![\mathsf{e}]\!]_\Gamma & \\
\mathcal{F}[\![\lambda(\mathsf{x} : \sigma).\, \mathsf{e}]\!]_\Gamma & & \\
\quad = \lambda(x :: \sigma).(\mathcal{F}[\![\mathsf{e}]\!]_{(\Gamma,\mathsf{x}:\sigma)} :: \sigma_2) & & \text{iff } \Gamma \vdash_{\mathsf{F}} \mathsf{e} : \sigma_2 \wedge \sigma_2 \in \mathcal{Q} \\
\quad = \lambda(x :: \sigma).\mathcal{F}[\![\mathsf{e}]\!]_{(\Gamma,\mathsf{x}:\sigma)} & & \text{otherwise} \\
\mathcal{F}[\![\mathsf{e}_1\ \mathsf{e}_2]\!]_\Gamma & & \\
\quad = \mathcal{F}[\![\mathsf{e}_1]\!]_\Gamma\ (\mathcal{F}[\![\mathsf{e}_2]\!]_\Gamma :: \sigma_2) & & \text{iff } \Gamma \vdash_{\mathsf{F}} \mathsf{e}_2 : \sigma_2 \wedge \sigma_2 \in \mathcal{Q} \\
\quad = \mathcal{F}[\![\mathsf{e}_1]\!]_\Gamma\ \mathcal{F}[\![\mathsf{e}_2]\!]_\Gamma & & \text{otherwise}
\end{array}$$

Figure 4. System F to HMF translation

Nevertheless, HMF extended with rigid type annotations can express any System F program since rigid annotations allow polymorphic values to be returned from a function. Figure 4 defines a translation function $\mathcal{F}[\![\mathsf{e}]\!]_\Gamma$ that translates any System F term e under a type environent Γ to a well-typed HMF term e. Note that $\mathcal{Q}$ denotes the set of quantified types and $\sigma \in \mathcal{Q}$ implies that $\sigma \neq \rho$ for any ρ. The expression $\Gamma \vdash_{\mathsf{F}} \mathsf{e} : \sigma$ states that the System F term e has type σ under a type environment Γ and is standard.

To translate a System F term to HMF, we keep variables untranslated and remove all type abstractions and applications. Parameters of a lambda expressions are kept annotated in the translated HMF term. If the body has a polymorphic type in the System F term, we also annotate the body in the HMF term since HMF cannot derive polymorphic types for unannotated lambda bodies. Applications are annotated whenever the argument is a quantified type.

There are of course other translations possible, and in many cases one can do with fewer annotations in practice. Nevertheless, the above translation is straightforward and removes most of the annotations that can be inferred automatically.

Theorem 2 (*Embedding of System F*):

If $\Gamma \vdash_{\mathsf{F}} \mathsf{e} : \sigma$ then $\Gamma \vdash \mathcal{F}[\![\mathsf{e}]\!]_\Gamma : \sigma'$ where $\sigma' \sqsubseteq \sigma$

6. Type inference

The type inference algorithm for HMF is a relatively small extension of algorithm W (Damas and Milner 1982) with subsumption and unification of quantified types. We first discuss unification and subsumption before describing the actual type inference algorithm.

6.1 Unification

Figure 5 describes a unification algorithm between polymorphic types. The algorithm is equivalent to standard Robinson unification (Robinson 1965) except that type variables can unify with polytypes and there is an extra case for unifying quantified types. The unification algorithm assumes that the types are in *normal form*. A type σ is in normal form when all quantifiers are bound and ordered with respect to their occurrence in the type. For example, $\forall\alpha\beta.\, \alpha \to \beta$ is in normal form, but $\forall\beta\alpha.\, \alpha \to \beta$ or $\forall\alpha.\, \mathit{Int}$ are not. Implementation wise, it is easy to keep types in normal form by returning the free variables of a type always in order of occurrence.

Having types in normal form makes it easy to unify quantified types. In the last case of *unify*, we replace the quantifiers of each type with fresh skolem constants *in order*, and unify the resulting unquantified types. Afterwards, we check that none of the skolems escape through a free variable which would be unsound. For example, if β is a free variable, we need to reject the unification of

```
$unify :: (\sigma_1, \sigma_2) \rightarrow S$
  where $\sigma_1$ and $\sigma_2$ are in normal form

$unify(\alpha, \alpha) =$
  return $[\,]$

$unify(\alpha, \sigma)$ or $unify(\sigma, \alpha) =$
  fail if $(\alpha \in ftv(\sigma))$        (‘occurs’ check)
  return $[\alpha := \sigma]$

$unify(c\ \sigma_1 \dots \sigma_n, c\ \sigma'_1 \dots \sigma'_n) =$
  let $S_1 = [\,]$
  let $S_{i+1} = unify(S_i\sigma_i, S_i\sigma'_i) \circ S_i$  for $i \in 1 \dots n$
  return $S_{n+1}$

$unify(\forall\alpha.\, \sigma_1, \forall\beta.\, \sigma_2) =$
  assume $c$ is a fresh (skolem) constant
  let $S = unify([\alpha := c]\sigma_1, [\beta := c]\sigma_2)$
  fail if $(c \in con(codom(S)))$       (‘escape’ check)
  return $S$
```

Figure 5. Unification

```
$subsume :: (\sigma_1, \sigma_2) \rightarrow S$
  where $\sigma_1$ and $\sigma_2$ are in normal form

$subsume(\forall\overline{\alpha}.\, \rho_1, \forall\overline{\beta}.\, \rho_2) =$
  assume $\overline{\beta}$ are fresh, and $\overline{c}$ are fresh (skolem) constants
  let $S = unify([\overline{\alpha} := \overline{c}]\rho_1, \rho_2)$
  fail if not $(\overline{c} \not\cap con(codom(S - \overline{\beta})))$     (‘escape’ check)
  return $(S - \overline{\beta})$
```

Figure 6. Subsumption

$\forall\alpha.\, \alpha \rightarrow \alpha$ and $\forall\alpha.\, \alpha \rightarrow \beta$. This check is done by ensuring that the codomain of the substitution does not contain the skolem constant c, and the unification fails if c is an element of $con(codom(S))$ (where $con(\cdot)$ returns the skolem constants in the codomain).

Theorem 3 (*Unification is sound*): If $unify(\sigma_1, \sigma_2) = S$ then $S\sigma_1 = S\sigma_2$.

Theorem 4 (*Unification is complete and most general*): If $S\sigma_1 = S\sigma_2$ then $unify(\sigma_1, \sigma_2) = S'$ where $S = S'' \circ S'$ for some S''.

6.2 Subsumption

Figure 6 defines subsumption where $subsume(\sigma_1, \sigma_2)$ returns a most general substitution S such that $S\sigma_2 \sqsubseteq S\sigma_1$. Informally, it instantiates σ_2 such that it can unify with the (potentially polymorphic) type σ_1. It uses the same mechanism that is usually used to implement the subsumption relation in type systems based on type containment (Odersky and Läufer 1996; Peyton Jones et al. 2007).

As shown in Figure 6, the algorithm first skolemizes the quantifiers of σ_1 and instantiates the quantifiers $\overline{\beta}$ of σ_2 with fresh type variables. Afterwards, we check that no skolems escape through free variables which would be unsound. For example, $subsume(\forall\alpha.\, \alpha \rightarrow \alpha, \forall\alpha\beta.\, \alpha \rightarrow \beta)$ succeeds, but it would be wrong to accept $subsume(\forall\alpha.\, \alpha \rightarrow \alpha, \forall\alpha.\, \alpha \rightarrow \beta)$ where β is a free variable. Note that in contrast with unification, we first remove the quantifiers $\overline{\beta}$ from the domain of the substitution since it is fine for those variables to unify with the skolems $\overline{c}$.

Theorem 5 (*Subsumption is sound*): If $subsume(\sigma_1, \sigma_2) = S$ then $S\sigma_2 \sqsubseteq S\sigma_1$.

```
$infer :: (\Gamma, e) \rightarrow (\theta, \sigma)$

$infer(\Gamma, x) =$
  return $([\,], \Gamma(x))$

$infer(\Gamma,$ let $x = e_1$ in $e_2) =$
  let $(\theta_1, \sigma_1) = infer(\Gamma, e_1)$
  let $(\theta_2, \sigma_2) = infer((\theta_1\Gamma, x : \sigma_1), e_2)$
  return $(\theta_2 \circ \theta_1, \sigma_2)$

$infer(\Gamma, \lambda x.e) =$
  assume $\alpha$ and $\overline{\beta}$ are fresh
  let $(\theta, \forall\overline{\beta}.\, \rho) = infer((\Gamma, x : \alpha), e)$
  return $(\theta, generalize(\theta\Gamma, \theta(\alpha \rightarrow \rho)))$

$infer(\Gamma, \lambda(x :: \exists\overline{\alpha}.\, \sigma).e) =$
  assume $\overline{\alpha}$ and $\overline{\beta}$ are fresh
  let $(\theta, \forall\overline{\beta}.\, \rho) = infer((\Gamma, x : \sigma), e)$
  return $(\theta, generalize(\theta\Gamma, \theta(\sigma \rightarrow \rho)))$

$infer(\Gamma, e_1\ e_2) =$
  assume $\overline{\alpha}$ are fresh
  let $(\theta_0, \forall\overline{\alpha}.\, \rho)$     $= infer(\Gamma, e_1)$
  let $(\theta_1, \sigma_1 \rightarrow \sigma) = funmatch(\rho)$
  let $(\theta_2, \sigma_2)$  $= infer(\theta_1\theta_0\Gamma, e_2)$
  let $(\Theta_3, \theta_3) = split(subsume(\theta_2\sigma_1, \sigma_2))$
  let $\theta_4$          $= \theta_3 \circ \theta_2 \circ \theta_1 \circ \theta_0$
  fail if not $(dom(\Theta_3) \not\cap ftv(\theta_4\Gamma))$
  return $(\theta_4, generalize(\theta_4\Gamma, \Theta_3\theta_4\sigma))$
```

Figure 7. Type inference for Plain HMF

```
$funmatch(\sigma_1 \rightarrow \sigma_2) =$
  return $([\,], \sigma_1 \rightarrow \sigma_2)$
$funmatch(\alpha) =$
  assume $\beta_1$ and $\beta_2$ are fresh
  return $([\alpha := \beta_1 \rightarrow \beta_2], \beta_1 \rightarrow \beta_2)$

$generalize(\Gamma, \sigma) =$
  let $\overline{\alpha} = ftv(\sigma) - ftv(\Gamma)$
  return $\forall\overline{\alpha}.\, \sigma$

$split(S) =$
  let $\theta_1$  $= [\alpha := \sigma \mid (\alpha := \sigma) \in S \wedge \sigma \in \mathcal{T}]$
  let $\Theta_1 = [\alpha := \sigma \mid (\alpha := \sigma) \in S \wedge \sigma \notin \mathcal{T}]$
  return $(\Theta_1, \theta_1)$
```

Figure 8. Helper functions

Theorem 6 (*Subsumption is partially complete and most general*): If $S\sigma_2 \sqsubseteq S\sigma_1$ holds and σ_1 is not a type variable, then $subsume(\sigma_1, \sigma_2) = S'$ where $S = S'' \circ S'$ for some S''.

If σ_1 is a type variable, we have that $subsume(\alpha, \forall\overline{\beta}.\, \rho)$ equals $[\alpha := \rho]$ for some fresh $\overline{\beta}$. When matching arguments to functions with a type of the form $\forall\alpha. \dots \rightarrow \alpha \rightarrow \dots$ this is exactly the disambiguating case that prefers predicative instantiation and a minimal polymorphic weight, and the reason why subsumption is only partially complete.

6.3 A type inference algorithm

Figure 7 defines a type inference algorithm for HMF. Given a type environment Γ and expression e, the function $infer(\Gamma, e)$ returns a monomorphic substitution θ and type σ such that σ is the principal type of e under $\theta\Gamma$.

In the inference algorithm we use the notation $\sigma \in \mathcal{T}$ when σ is a monomorphic type, i.e. $\sigma = \tau$. The expression $\sigma \notin \mathcal{T}$ is used for polymorphic types when there exist no τ such that $\sigma = \tau$. We use the notation θ for monomorphic substitutions, where $\sigma \in codom(\theta)$ implies $\sigma \in \mathcal{T}$, and the notation Θ for polymorphic substitutions where $\sigma \in codom(\Theta)$ implies $\sigma \notin \mathcal{T}$. The function $split(S)$ splits any substitution S into two substitutions θ and Θ such that $S = \Theta \circ \theta$.

The rules for variables and **let** expressions are trivial. In the rules for lambda expressions, we first instantiate the result type of the body and than generalize over the function type. For unannotated parameters, we can assume a fresh type α in the type environment while annotated parameters get their given type.

The application rule is more involved but still very similar to the usual application rule in algorithm W (Damas and Milner 1982). Instead of unifying the argument with the parameter type, we use the $subsume$ operation since we may need to instantiate the argument type. The polymorphic substitution S returned from $subsume$ is split in a monomorphic substitution θ_3 and a polymorphic substitution Θ_3, such that $S = \Theta_3 \circ \theta_3$. Next, we check that no polymorphic types escape through free variables in the type environment by ensuring that $dom(\Theta_3) \not\cap ftv(\theta_4\Gamma)$. This is necessary since rule FUN can only assume monotypes τ for parameters, and without the check we would be able to infer polymorphic types for parameters. Since the domain of Θ_3 does not occur in the type environment, we can apply the polymorphic substitution to the result type, and return the generalized result together with a monomorphic substitution.

We can now state our main theorems that type inference for (Plain) HMF is sound and complete:

Theorem 7 (*Type inference is sound*): If $infer(\Gamma, e) = (\theta, \sigma)$ then $\theta\Gamma \vdash e : \sigma$ holds.

Theorem 8 (*Type inference is complete and principal*): If $\theta\Gamma \vdash e : \sigma$, then $infer(\Gamma, e) = (\theta', \sigma')$ where $\theta \approx \theta'' \circ \theta'$ and $\theta''\sigma' \sqsubseteq \sigma$.

Following Jones (1995), we use the notation $S_1 \approx S_2$ to indicate that $S_1\alpha = S_2\alpha$ for all but a finite number of fresh type variables. In most cases, we can treat $S_1 \approx S_2$ as $S_1 = S_2$ since the only differences between substitutions occur at variables which are not used elsewhere in the algorithm. We need this mechanism because the algorithm introduces fresh variables that do not appear in the hypotheses of the rule or other distinct branches of the derivation.

6.4 Optimizations

In practice, inference algorithms tend to use direct updateable references instead of using an explicit substitution. This works well with HMF too, but certain operations on substitutions must be avoided. When unifying quantified types in the $unify$ algorithm, the check $(c \in con(codom(S)))$ can be implemented more effectively when using references as $(c \in con(S(\forall\alpha.\,\sigma_1)) \cup con(S(\forall\beta.\,\sigma_2))$ (and similarly in $subsume$).

In the application case of $infer$, we both $split$ the substitution and there is a check that $(dom(\Theta_3) \not\cap ftv(\theta_4\Gamma))$ which ensures that no poly type escapes into the environment. However, since let-bound values in the environment always have a generalized type, the only free type variables in the environment are introduced by lambda-bound parameter types. Therefore, the check can be delayed, and done instead when checking lambda expressions. Effectively, we remove the $split$ and move the check from the application rule to the lambda case:

```
$infer(\Gamma, \lambda x.e) =$
  assume $\alpha$ and $\overline{\beta}$ are fresh
  let $(S, \forall\overline{\beta}.\,\rho) = infer((\Gamma, x : \alpha), e)$
  fail if $(S\alpha \notin \mathcal{T})$
  return $(S,\ generalize(S\Gamma, S(\alpha \to \rho)))$
```

This change makes it directly apparent that only monomorphic types are inferred for lambda bound parameters. Of course, it also introduces polymorphic substitutions everywhere, but when using an updateable reference implementation this happens anyway. Note that this technique can actually also be applied in higher-rank inference systems (Peyton Jones et al. 2007; Odersky and Läufer 1996) removing the 'escaping skolem' check in subsumption.

6.5 Rigid annotations

It is straightforward to extend the type inference algorithm with rigid type annotations, since expressions can be checked syntactically if they are annotated or not. In the application case of the algorithm specified in Figure 7, we use $unify$ instead of $subsume$ whenever the argument expression e_2 is annotated, which effectively prevents the instantiation of the argument type. Finally, we adapt the case for lambda expressions to not instantiate the type of an annotated body.

6.6 N-ary applications

Implementing inference that disambiguates over multiple arguments using rule APP-N is more involved. First we need to extend subsumption to work on multiple arguments at once:

```
$subsumeN(\sigma_1 \ldots \sigma_n, \sigma'_1 \ldots \sigma'_n) =$
  let $i\ = $ if $\sigma_i \in \{\sigma_1, \ldots, \sigma_n\} \wedge \sigma_i \notin \mathcal{V}$ then $i$ else $1$
  let $S = subsume(\sigma_i, \sigma'_i)$
  if $n = 1$ then return $S$
    else return $S \circ subsumeN(S(\sigma_1 \ldots \sigma_{i-1}\ \sigma_{i+1} \ldots \sigma_n),$
                                  $S(\sigma'_1 \ldots \sigma'_{i-1}\ \sigma'_{i+1} \ldots \sigma'_n))$
```

The function $subsumeN$ applies subsumption to n parameter types $\sigma_1 \ldots \sigma_n$ with the supplied argument types $\sigma'_1 \ldots \sigma'_n$. Due to sharing, we can often infer a polymorphic type after matching some arguments, as happens for example in $revapp\ id\ poly$ where the $poly$ argument is matched first. The trick is now to subsume the parameter and argument pairs in the right order to disambiguate correctly. Since subsumption is unambiguous for parameter types that are not a type variable ($\sigma_i \notin \mathcal{V}$), we first pick these parameter types. Only when such parameters are exhausted, we subsume the rest of the parameters, where the order does not matter and we arbitrarily pick the first. In a previous version of the system, we subsumed in order of dependencies between parameter and argument types, but one can show that this is unnecessary – if there is any type variable shared between parameter and argument types, it must be (lambda) bound in the environment, and in that case, we cannot infer a polymorphic type regardless of the order of subsumption.

Secondly, we extend function matching to return as many known parameter types as possible, where we pass the number of supplied arguments n:

```
$funmatchN(n, \sigma_1 \to \ldots \to \sigma_m \to \sigma) =$
  where $m$ is the largest possible with $1 \leqslant m \leqslant n$
  return $([\,],\ \sigma_1 \ldots \sigma_m,\ \sigma)$
$funmatchN(n, \alpha) =$
  assume $\beta_1$ and $\beta_2$ are fresh
  return $([\alpha := \beta_1 \to \beta_2],\ \beta_1,\ \beta_2)$
```

During inference, we now consider all arguments at once, where we first infer the type of the function, and then call the helper function $inferapp$ with the found type:

```
$infer(\Gamma, e\ e_1 \ldots e_n) =$
  assume $n$ is the largest possible with $n \geqslant 1$
  let $(\theta_1, \sigma_1) = infer(\Gamma, e)$
```

```
    let $(\theta_2, \sigma_2) = \mathit{inferapp}(\theta_1\Gamma, \sigma_1, e_1 \ldots e_n)$
    return $(\theta_2 \circ \theta_1,\ \sigma_2)$
```

The *inferapp* function is defined separately as it calls itself recursively for each polymorphic function result until all n arguments are consumed:

```
$\mathit{inferapp}(\Gamma, \forall\overline{\alpha}.\,\rho, e_1 \ldots e_n) =$
    assume $\overline{\alpha}$ is fresh and $n \geqslant 1$
    let $(\theta_0, \sigma_1 \ldots \sigma_m, \sigma) = \mathit{funmatchN}(n, \rho)$
    let $(\theta'_i, \sigma'_i) = \mathit{infer}(\theta_{i-1}\Gamma, e_i)$    for $1 \leqslant i \leqslant m$
    let $\theta_i$         $= \theta'_i \circ \theta_{i-1}$
    let $(\Theta, \theta') = \mathit{split}(\mathit{subsumeN}(\theta_m(\sigma_1 \ldots \sigma_m),$
                                        $\theta_m(\sigma'_1 \ldots \sigma'_m)))$
    let $\theta$          $= \theta' \circ \theta_m$
    fail if not $(\mathit{dom}(\Theta) \not\cap \mathit{ftv}(\theta\Gamma))$
    if $m < n$ then return $\mathit{inferapp}(\theta\Gamma, \Theta\theta\sigma, e_{m+1} \ldots e_n)$
               else return $(\theta,\ \mathit{generalize}(\theta\Gamma, \Theta\theta\sigma))$
```

First, *funmatchN* is used to consider as many arguments m as possible. Note that m is always smaller or equal to n. Next, the types of the next m arguments are inferred, and the *subsumeN* function applies subsumption to all the parameter types with the found argument types. Afterwards we check again that no polymorphic types escape in the environment. Finally, if there are still arguments left (as in $\mathit{head}\ \mathit{ids}\ 1$ for example), *inferapp* is called recursively with the remaining arguments and the found result type. Otherwise, the generalized result type is returned.

7. Related work

In Section 3 we already discussed MLF and boxy type inference. MLF was first described by by Rémy and Le Botlan (2004; 2003; 2007; 2007). The extension of MLF with qualified types is described in (Leijen and Löh 2005). Leijen later gives a type directed translation of MLF to System F and describes Rigid-MLF (Leijen 2007c), a variant of MLF that does not assign polymorphically bounded types to let-bound values but internally still needs the full inference algorithm of MLF.

Vytiniotis et al. (2006) describe boxy type inference which is made principal by distinguishing between inferred 'boxy' types, and checked annotated types. A critique of boxy type inference is that its specification has a strong algorithmic flavor which can make it fragile under small program transformations (Rémy 2005).

Recently, Vytiniotis et al. (2008) presented new version of boxy type inference called FPH. The system has a simple annotation rule where polymorphic parameters require an annotation, and where let-bindings with a higher-rank type may require an annotation. An advantage of using boxy types, is that FPH does not need minimality conditions like HMF. A drawback though is that FPH cannot always assign a principal type to an expression. Also, it shares with boxy type inference that some common expressions with unambiguous typings are still rejected (see Section 3). For example, suppose we add the *id* function to a list of polymorphic identity functions:

let $xs = \mathit{cons}\ \mathit{id}\ \mathit{ids}$

where *cons* has type $\forall\alpha.\,\alpha \to [\alpha] \to [\alpha]$. The above definition is rejected in FPH even though the result type, $[\forall\alpha.\,\alpha \to \alpha]$, is unambiguous (and inferred by HMF).

To the best of our knowledge, a type inference algorithm for the simply typed lambda calculus was first described by Curry and Feys (1958). Later, Hindley (1969) introduced the notion of principal type, proving that the Curry and Feys algorithm inferred most general types. Milner (1978) independently described a similar algorithm, but also introduced the important notion of first-order polymorphism where let-bound values can have a polymorphic type. Damas and Milner (1982) later proved the completeness of Milner's algorithm, extending the type inference system with polymorphic references (Damas 1985). Wells (1999) shows that general type inference for unannotated System F is undecidable.

Jones (1997) extends Hindley-Milner with first class polymorphism by wrapping polymorphic values into type constructors. This is a simple and effective technique that is widely used in Haskell but one needs to define a special constructor and operations for every polymorphic type. Garrigue and Rémy (1999) use a similar technique but can use a generic 'box' operation to wrap polymorphic types. Odersky and Läufer (1996) describe a type system that has higher-rank types but no impredicative instantiation. Peyton Jones et al. (2007) extend this work with type annotation propagation. Dijkstra (2005) extends this further with bidirectional annotation propagation to support impredicative instantiation.

8. Future work

We feel that both HMF and MLF present interesting points in the design space of type inference for first-class polymorphism. (Full) MLF is an upper bound: it is the most expressive system to date, requiring only annotations on parameters that are used polymorphically, but it also introduces more complexity with the introduction of polymorphically bounded types. The lower bound in the design space is represented by HMF, which uses just System F types, but also requires annotations on ambiguous impredicative applications.

Currently, we are working on a third system, called HML, that resides between these design points (Leijen 2008). This system is a simplification of MLF that only uses flexible types. The addition of flexible quantification leads to a very simple annotation rule where only function parameters with a polymorphic type need an annotation, but it still retains much of the expressiveness of MLF.

9. Conclusion

HMF is a conservative extention of Hindley-Milner type inference that supports first-class polymorphism and has an effective type inference algorithm that is just a small modification of algorithm W. Given the relative simplicity combined with expressive power, we feel that this system can be a great candidate as the basic type system for future languages that want to support first-class polymorphic programming with minimal implementation effort.

Acknowledgements

The author would like to thank Dimitrios Vytiniotis and Simon Peyton Jones for their feedback on an earlier version of the type rules, and to Didier Remy who suggested the extension of HMF to N-ary applications. Thanks also to Edsko de Vries for his feedback and for implementing a full version of the HMF inference algorithm in the Morrow compiler.

References

H. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.

Luis Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, April 1985. Technical report CST-33-85.

Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th ACM symp. on Principles of Programming Languages (POPL'82)*, pages 207–212, 1982.

Atze Dijkstra. *Stepping through Haskell*. PhD thesis, Universiteit Utrecht, Nov. 2005.

Jacques Garrigue and Didier Rémy. Semi-explicit first-class polymorphism for ML. *Journal of Information and Computation*, 155:134–169, 1999.

J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, Dec. 1969.

Mark P. Jones. First-class polymorphism with type inference. In *24th ACM Symposium on Principles of Programming Languages (POPL'97)*, January 1997.

Mark P. Jones. Formal properties of the Hindley-Milner type system. Unpublished notes, August 1995.

Didier Le Botlan. *ML^F: Une extension de ML avec polymorphisme de second ordre et instanciation implicite*. PhD thesis, INRIA Rocquencourt, May 2004. Also in English.

Didier Le Botlan and Didier Rémy. MLF: Raising ML to the power of System-F. In *The International Conference on Functional Programming (ICFP'03)*, pages 27–38, aug 2003.

Didier Le Botlan and Didier Rémy. Recasting MLF. Research Report 6228, INRIA, Rocquencourt, France, June 2007.

Daan Leijen. A reference implementation of HMF. Available at `http://research.microsoft.com/users/daan/pubs.html`, September 2007a.

Daan Leijen. HMF: Simple type inference for first-class polymorphism. Technical Report MSR-TR-2007-118, Microsoft Research, September 2007b. Extended version with proofs.

Daan Leijen. Flexible types: robust type inference for first-class polymorphism. Technical Report MSR-TR-2008-55, Microsoft Research, March 2008.

Daan Leijen. A type directed translation from MLF to System F. In *The International Conference on Functional Programming (ICFP'07)*, Oct. 2007c.

Daan Leijen and Andres Löh. Qualified types for MLF. In *The International Conference on Functional Programming (ICFP'05)*. ACM Press, Sep. 2005.

Xavier Leroy and M Mauny. Dynamics in ML. In *ACM conference on Functional Programming and Computer Architecture (FPCA'91)*. Springer-Verlag, 1991. volume 523 of LNCS.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:248–375, 1978.

Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *23th ACM symp. on Principles of Programming Languages (POPL'96)*, pages 54–67, January 1996.

Simon Peyton Jones and Mark Shields. Lexically scoped type variables. Draft, March 2004.

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

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

Didier Rémy. Simple, partial type-inference for System-F based on type-containment. In *The International Conference on Functional Programming (ICFP'05)*, September 2005.

Didier Rémy and Boris Yakobowski. A graphical presentation of MLF types with a linear-time unification algorithm. In *TLDI'07*, pages 27–38, 2007.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: type inference for higher-rank types and impredicativity. In *The International Conference on Functional Programming (ICFP'06)*, September 2006.

Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. FPH : First-class polymorphism for Haskell. In *13th ACM symp. of the International Conference on Functional Programming (ICFP'08)*, September 2008.

J.B. Wells. Typability and type checking in System-F are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1–3):111–156,1999.