# Flexible types

## Robust type inference for first-class polymorphism

Daan Leijen
Microsoft Research
daan@microsoft.com

## Abstract

We present HML, a type inference system that supports full first-class polymorphism where few annotations are needed: only function parameters with a polymorphic type need to be annotated. HML is a simplification of MLF where only flexibly quantified types are used. This makes the types easier to work with from a programmers perspective, and we found that it simplifies the implementation of the type inference algorithm. Still, HML retains much of the expressiveness of MLF, it is robust with respect to small program transformations, and has a simple specification of the type rules with an effective type inference algorithm that infers principal types. A reference implementation of the type system is available at: `http://research.microsoft.com/users/daan/pubs.html`.

## 1. Introduction

Most type inference systems used in practice are based on Hindley-Milner type inference (Hindley 1969; Milner 1978; Damas and Milner 1982). This is an attractive type system since it has a simple logical specification, and an effective type inference algorithm that can automatically infer most general, or *principal*, types for expressions without any further type annotations. Unfortunately, Hindley-Milner inference does not support first-class polymorphic values, and only allows a simple form of polymorphism on let-bound values. This is a severe restriction 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 practical.

There is a long list of research papers that propose type inference systems for System F (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; Leijen 2007a). All of these systems mostly differ in where type annotations are needed in the program, and how easy it is for the programmer to determine where to put them in practice.

The MLF type inference system (Le Botlan and Rémy 2003; Le Botlan 2004) is the most expressive inference system to date, and requires type annotations on function parameters that are used at two or more polymorphic types. Despite its good properties, MLF has found little acceptance in practice, perhaps partly due to the intricate type structure with both flexible and rigid quantification, and a more involved type inference algorithm.

In this article, we present a simplification of MLF, called HML, that uses only flexible types. This simplifies both the types and the inference algorithm, while retaining many of the good properties of MLF. In particular:

- HML is a simplification and restriction of MLF, and has exactly the same expressiveness of Implicit MLF (Le Botlan and Rémy 2007b). A more conservative annotation rule leads to type rules that only need flexible types and avoid rigidly quantified types completely. This makes types easier to work with from a programmers perspective, and we show that it also simplifies the type inference algorithm.
- It is important to have a clear rule where and when annotations are needed. HML has a very simple annotation rule: only function parameters with a polymorphic type need to be annotated.
- As a consequence, HML is a conservative extension of Hindley-Milner: every program that is well-typed in Hindley-Milner is also a well-typed HML program and type annotations are never required for such programs.
- The type rules of HML are very similar to MLF, where flexible types enable principal derivations where every expression can be assigned a most general type. This allows a programmer to use modular reasoning about a program, and enables efficient type inference.
- HML is very robust with respect to small program transformations. It has the property that whenever the application $e_1\ e_2$ is well-typed, so is the abstraction $\mathit{apply}\ e_1\ e_2$ and reverse application $\mathit{revapp}\ e_2\ e_1$. Moreover, we can always inline a let-binding, or abstract a common expression into a shared let-binding. We consider this an important property as it forms the basis of equational reasoning where we expect to be able to replace equals by equals.

In the following section we give an overview of HML in practice. Section 4 presents the formal logical type rules of HML. In Section 7.2 we discuss a restriction where annotations only consist of System F types. Finally, Section 8 and Section 9 describe the unification and type inference algorithm.

[Microsoft Research Technical Report: MSR-TR-2008-55, March 2008]

## 2. An overview

Flexible types gives us a powerful type inference system for programs with full first-class polymorphism where types can be of higher-rank and impredicatively instantiated. Unfortunately, first-class polymorphic inference is undecidable in general (Wells 1999), and some type annotations are always needed. Previously proposed type inference systems that support first-class polymorphism mostly differ where such annotations are needed in the program, and how easy it is for the programmer to apply the those rules in practice. HML has a very simple annotation rule:

*function parameters with a polymorphic type need an annotation*

and that's it! In practice this means that an annotation is only needed when definining polymorphic parameters. Take for example the following function:

$$poly = \lambda(f :: \forall\alpha.\, \alpha \to \alpha).\, (f\ 1, f\ \mathit{True})$$

The parameter $f$ of the $poly$ function must be annotated with a polymorphic type since it is applied to both an integer and a boolean value. In general, one can never infer a type for such parameter since there are many possible types for $f$: for example $\forall\alpha.\, \alpha \to \alpha \to \alpha$ or $\forall\alpha.\, \alpha \to \mathit{Int}$. Therefore HML simply requires annotations on all parameters with a polymorphic type.

These are the only annotations that are ever required, and polymorphic functions and data structures can be used freely without further annotations. Assuming the following definitions:

$$\begin{array}{ll} id & :: \forall\alpha.\, \alpha \to \alpha \\ apply & :: \forall\alpha\beta.\, (\alpha \to \beta) \to \alpha \to \beta \\ revapp & :: \forall\alpha\beta.\, \alpha \to (\alpha \to \beta) \to \beta \end{array}$$

HML can type $poly\ \ id$ or $poly\ \ (\lambda x.\, x)$ for example. Impredicative instantiations are also inferred automatically. For example $apply\ poly\ id$, or $revapp\ id\ poly$, are accepted where the $\alpha$ quantifier of $apply$ and $revapp$ is instantiated to the polymorphic type $\forall\alpha.\, \alpha \to \alpha$. In general, HML is not sensitive to the order of applications and whenever the application $e_1\ \ e_2$ is accepted, so is $apply\ \ e_1\ \ e_2$ and $revapp\ \ e_2\ \ e_1$. This is an important property in practice since it allows us to apply the usual abstractions uniformly over polymorphic values too.

### 2.1 Flexible types

Type inference works well with data structures with polymorphic elements too. Assuming:

$$\begin{array}{ll} inc & :: \mathit{Int} \to \mathit{Int} \\ single & :: \forall\alpha.\, \alpha \to \mathit{List}\, \alpha \\ append & :: \forall\alpha.\, \mathit{List}\, \alpha \to \mathit{List}\, \alpha \to \mathit{List}\, \alpha \\ map & :: \forall\alpha\beta.\, (\alpha \to \beta) \to \mathit{List}\, \alpha \to \mathit{List}\, \beta \end{array}$$

we can for example map the $poly$ function over a list of polymorphic identity functions as $map\ poly\ (single\ id)$, where $single\ id$ has type $\mathit{List}\, (\forall\alpha.\, \alpha \to \alpha)$. Of course, sometimes a monomorphic type is inferred the expression $single\ \ id$. Take for example $append\ \ (single\ \ inc)\ \ (single\ \ id)$ where $single\ id$ must get the type $\mathit{List}\, (\mathit{Int} \to \mathit{Int})$. We can wonder now what happens when we share the $single\ id$ expression, as in the following program:

$$\begin{array}{l} \mathbf{let}\ ids = single\ id \\ \mathbf{in}\ \ (map\ poly\ ids, append\ (single\ inc)\ ids) \end{array}$$

Of course, according to our annotation rule, this program is accepted as is. This implies that we must be able to use $ids$ both as a list of polymorphic elements, $\mathit{List}\ (\forall\alpha.\, \alpha \to \alpha)$, and as a list of integer functions, $\mathit{List}\ (\mathit{Int} \to \mathit{Int})$, even though these types are incomparable in System F.

To address this, HML uses *flexible types* to assign a most general type to $ids$, namely $\forall(\beta \geqslant \forall\alpha.\, \alpha \to \alpha).\, \mathit{List}\, \beta$. We can read this as "for any type $\beta$ that is an instance of $\forall\alpha.\, \alpha \to \alpha$, this is a list of $\beta$", and we can instantiate this type to both of the above types, i.e.:

$$\begin{array}{lcl} \forall(\beta \geqslant \forall\alpha.\, \alpha \to \alpha).\, \mathit{List}\, \beta & \sqsubseteq & \mathit{List}\, (\forall\alpha.\, \alpha \to \alpha) \\ \forall(\beta \geqslant \forall\alpha.\, \alpha \to \alpha).\, \mathit{List}\, \beta & \sqsubseteq & \forall\alpha.\, \mathit{List}\, (\alpha \to \alpha) \\ \forall(\beta \geqslant \forall\alpha.\, \alpha \to \alpha).\, \mathit{List}\, \beta & \sqsubseteq & \mathit{List}\, (\mathit{Int} \to \mathit{Int}) \\ \ldots & & \end{array}$$

Flexible types are key to enable modular type inference where every expression can be assigned a most general, or principal, type. By putting the bound on the quantifier, flexible types also neatly keep track shared polymorphic types. Take for example the $choose$ function:

$$choose :: \forall\alpha.\, \alpha \to \alpha \to \alpha$$

The expression $choose\ \ id$ can be assigned two incomparable types in System F, namely $\forall\alpha.\, (\alpha \to \alpha) \to \alpha \to \alpha$ or $(\forall\alpha.\, \alpha \to \alpha) \to (\forall\alpha.\, \alpha \to \alpha)$. In HML, we can assign a principal type to this expression, namely $\forall(\beta \geqslant \forall\alpha.\, \alpha \to \alpha).\, \beta \to \beta$ that can be instantiated to both of the above System F types.

### 2.2 Robustness

Flexible types make the system very robust under rewrites. We have seen for example the system is insensitive to the order of arguments: whenever $e_1\ \ e_2$ is accepted, so is $apply\ \ e_1\ \ e_2$ and $revapp\ \ e_2\ \ e_1$. Also, we can always abstract or inline a shared expression: whenever $\mathbf{let}\ \ x\ =\ e_1\ \ \mathbf{in}\ \ e_2$ is accepted, so is $[x := e_1]e_2$, and the other way around, where we share a common expression through a let-binding. We consider this an important property as it stands at the basis of equational reasoning where equals can be substituted for equals.

The system is not robust with regard to $\eta$-expansion though since all polymophic parameters must be annotated. For example we cannot $\eta$-expand $poly$ to $\lambda f.\, poly\ f$ since $f$ has a polymorphic type, and we need to write $\lambda(f :: \forall\alpha.\, \alpha \to \alpha).\, poly\ f$.

In HML, we only allow flexible types on let-bindings and cannot pass values of a flexible type as an argument. Therefore, we cannot replace let-bindings with a flexible type by lambda abstractions. This situation is similar in Hindley-Milner where only let-bindings can have a quantified type scheme and where lambda bindings are restricted to monomorphic types.

### 2.3 Implicit MLF

Implicit MLF is a restriction of MLF where flexible types cannot be assigned to function parameters (Le Botlan and Rémy 2007b). This restriction was introduced to make it possible to assign a semantic meaning to flexible types in terms of System F types. Implicit MLF has no type annotations at all though and type inference is undecidable for this system. Le Botlan and Rémy describe a decidable variant called Explicit MLF (XMLF) that needs annotations on all function parameters that are *used* at two or more polymorphic instances. Unfortunately, this annotation rule leads to the introduction of rigidly quantified types that complicate the type system and its inference algorithm (but does support $\eta$-expansion).

HML can be seen as another decidable variant of Implicit MLF, and has exactly the same expressiveness. The key difference is a slightly more conservative annotation rule than XMLF where we require that *all* polymorphic function parameters must be annotated. Surprisingly, this simplifies the type system significantly since we found that rigid bounds are now no longer required, and as a result, we can also simplify the type inference algorithm where we no longer need to compute polynomial weights over types.

In Section 7.2 we describe a restriction of HML, called Rigid HML, which disallows flexible types on let-bindings too, and requires annotations on let-bindings that have a flexible type. Even though more annotations are needed, this could perhaps be a use-

$$\begin{array}{lll}
\text{Types} & \sigma & ::= \forall\alpha.\,\sigma \\
 & & \mid \alpha \\
 & & \mid c\ \sigma_1\ \ldots\ \sigma_n \\[1ex]
\text{Type schemes} & \varphi & ::= \forall(\alpha \geqslant \hat{\varphi}_1).\,\varphi_2 \\
 & & \mid \sigma \\
 & & \mid \bot \\[1ex]
\text{Quantified types} & \hat{\varphi} & ::= \forall(\alpha \geqslant \hat{\varphi}_1).\,\varphi_2 \quad \text{with } \alpha \in \mathit{ftv}(\varphi_2) \\
 & & \mid \bot \\[1ex]
\text{Prefix} & Q & ::= \alpha_1 \geqslant \hat{\varphi}_1, \ldots, \alpha_n \geqslant \hat{\varphi}_n \\[1ex]
\text{Mono types} & \tau & ::= \alpha \mid c\ \tau_1\ \ldots\ \tau_n \\
\text{Unquantified types} & \rho & ::= \alpha \mid c\ \sigma_1\ \ldots\ \sigma_n \\[1ex]
\text{Syntactic sugar} & \forall\alpha & = \forall(\alpha \geqslant \bot) \\
 & \forall Q.\,\varphi & = \forall(\alpha_1 \geqslant \hat{\varphi}_1).\ \ldots \forall(\alpha_n \geqslant \hat{\varphi}_n).\,\varphi
\end{array}$$

**Figure 1.** Types.

$$\begin{array}{ll}
\textsc{Var} & \dfrac{x : \varphi \in \Gamma}{Q, \Gamma \vdash x : \varphi} \\[2ex]
\textsc{Inst} & \dfrac{Q, \Gamma \vdash e : \varphi_1 \quad Q \vdash \varphi_1 \sqsubseteq \varphi_2}{Q, \Gamma \vdash e : \varphi_2} \\[2ex]
\textsc{Gen} & \dfrac{(Q, \alpha \geqslant \hat{\varphi}_1), \Gamma \vdash e : \varphi_2 \quad \alpha \notin \mathit{ftv}(\Gamma)}{Q, \Gamma \vdash e : \forall(\alpha \geqslant \hat{\varphi}_1).\,\varphi_2} \\[2ex]
\textsc{App} & \dfrac{Q, \Gamma \vdash e_1 : \sigma_2 \to \sigma \quad Q, \Gamma \vdash e_2 : \sigma_2}{Q, \Gamma \vdash e_1\ e_2 : \sigma} \\[2ex]
\textsc{Let} & \dfrac{Q, \Gamma \vdash e_1 : \varphi_1 \quad Q, (\Gamma, x : \varphi_1) \vdash e_2 : \varphi_2}{Q, \Gamma \vdash\ \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \varphi_2} \\[2ex]
\textsc{Fun} & \dfrac{Q, (\Gamma, x : \tau) \vdash e : \sigma}{Q, \Gamma \vdash \lambda x.\, e : \tau \to \sigma} \\[2ex]
\textsc{Fun-Ann} & \dfrac{Q, (\Gamma, x : \sigma_1) \vdash e : \sigma_2}{Q, \Gamma \vdash \lambda(x :: \sigma_1).\, e : \sigma_1 \to \sigma_2}
\end{array}$$

**Figure 2.** Type rules.

ful simplification in practice as it removes flexible types from the realm of the programmer.

## 3. Types

The types of HML are essentially equivalent to those of Implicit MLF and are defined formally in Figure 1. Basic types $\sigma$ are equivalent to regular System F types and are either a type variable $\alpha$, a constructor application $c\ \sigma_1\ \ldots\ \sigma_n$, or a quantified type $\forall\alpha.\,\sigma$. We do not need to treat the function constructor $\to$ specially and assume it is part of the constructors $c$.

Lambda bound values always have a $\sigma$ type. In contrast, let bound values can have a flexible type or *type scheme* $\varphi$. Type schemes are either a System F type $\sigma$, the most polymorphic type bottom ($\bot$), or a flexible type $\forall(\alpha \geqslant \hat{\varphi}_1).\,\varphi_2$.

A quantifier with a flexible bound can be instantiated to any instance of its bound. In particular, a quantifier $\forall(\alpha \geqslant \bot)$ can be instantiated to any type since $\bot$ is the most polymorphic type. We call such bound an *unconstrained bound* and usually shorten it to $\forall\alpha$. As we see later, the $\bot$ type is equivalent to $\forall\alpha.\,\alpha$. Note that in System F, all quantifiers are always unconstrained, i.e. $\hat{\varphi}$ can only be $\bot$.

Implicit MLF allows arbitrary types in bounds where we can write types as $\forall(\alpha \geqslant \mathit{Int}).\,\alpha \to \alpha$ or $\forall(\beta \geqslant \mathit{List}\ (\forall\alpha.\,\alpha \to \alpha)).\,\beta$ where the bounds cannot be further instantiated. As it turns out, allowing such 'trivial' bounds leads to some unnecessary technical complications, and in HML we consider such types malformed, and the previous types should be written as $\mathit{Int} \to \mathit{Int}$ or $\mathit{List}\ (\forall\alpha.\,\alpha \to \alpha)$ respectively.

In particular, HML restricts types in a bound to non-trivial *quantified types* $\hat{\varphi}$ (which we denote using a hat ˆ symbol). A $\hat{\varphi}$ type is either bottom ($\bot$) or a quantified type $\forall(\alpha \geqslant \hat{\varphi}_1).\,\varphi_2$ where $\alpha \in \mathit{ftv}(\varphi_2)$. The side condition prevents the use of unbound quantifiers to hide a trivial bound, as in $\forall(\alpha \geqslant \forall\beta.\,\mathit{Int}).\,\alpha \to \alpha$ for example, and ensures that a bound can always be further instantiated.

A *prefix* $Q$ is a list of quantifiers $\alpha_1 \geqslant \hat{\varphi}_1, \ldots, \alpha_n \geqslant \hat{\varphi}_n$, where we assume that all the quantified type variables are distinct and form the domain of $Q$, written as $\mathit{dom}(Q)$, e.g. the set $\{\alpha_1, \ldots, \alpha_n\}$. If all quantifiers are unconstrained, we call $Q$ an *unconstrained prefix*.

We defined *monomorphic types* $\tau$ as equivalent to Hindley-Milner monomorphic types, and *unquantified types* $\rho$ as types without outer quantifiers. The free type variables of a type are defined in the usual way, where we take special care for types appearing in a bound:

$$\begin{array}{ll}
\mathit{ftv}(\bot) & = \varnothing \\
\mathit{ftv}(\alpha) & = \{\alpha\} \\
\mathit{ftv}(c\ \sigma_1\ \ldots\ \sigma_n) & = \mathit{ftv}(\sigma_1) \cup\ \ldots\ \cup \mathit{ftv}(\sigma_n) \\
\mathit{ftv}(\forall(\alpha \geqslant \hat{\varphi}_1).\,\varphi_2) & \\
\quad = \mathit{ftv}(\hat{\varphi}_1) \cup (\mathit{ftv}(\varphi_2) - \alpha) & \text{iff } \alpha \in \mathit{ftv}(\varphi_2) \\
\quad = \mathit{ftv}(\varphi_2) & \text{otherwise}
\end{array}$$

and this definition is extended in the natural way for sets, environments, and other structures containing types.

## 4. Type rules

The type rules of HML are defined in Figure 2. The expression $Q, \Gamma \vdash e : \varphi$ states that an expression $e$ can be assigned a type $\varphi$ under a type environment $\Gamma$ and prefix $Q$. The type environment $\Gamma$ binds variables to types where we write $\Gamma, x : \sigma$ to extend the environment $\Gamma$ with a new binding $x$ with type $\sigma$ (replacing any previous binding for $x$). The prefix $Q$ contains all the (flexible) bounds of the free type variables in $\Gamma$, $e$, and $\varphi$, and we have $(\mathit{ftv}(\Gamma) \cup \mathit{ftv}(e) \cup \mathit{ftv}(\varphi)) \subseteq \mathit{dom}(Q)$. In the definition of the Hindley-Milner type rules, the prefix $Q$ always contains unconstrained bounds ($\alpha \geqslant \bot$) and is therefore usually left implicit. The expression language is standard and defined as:

$$\begin{array}{ll}
e ::= x & \text{(variable)} \\
\quad\ \ e_1\ e_2 & \text{(application)} \\
\quad\ \ \lambda x.\, e & \text{(lambda expression)} \\
\quad\ \ \lambda(x :: \sigma).\, e & \text{(annotated lambda expression)} \\
\quad\ \ \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 & \text{(let-binding)}
\end{array}$$

Most rules are almost equivalent to the usual Hindley-Milner rules, except that all of them are stated under the prefix $Q$. The variable rule Var assigns the type that is found in the environment to a variable. The instance rule Inst states that we can always use an instance of a derived type, where we write $Q \vdash \varphi_1 \sqsubseteq \varphi_2$ to denote that $\varphi_2$ is an instance of $\varphi_1$ under a prefix $Q$. The generalization rule Gen moves an assumed bounded type from the prefix to the type, effectively generalizing over that type variable. The application rule App requires the argument and parameter type to be equivalent. Note that since the parameter type occurs under the arrow, the types in an application are $\sigma$ types (and not type schemes). Dually, the rule for let-bindings binds a generalized type scheme $\varphi$ to the binding variable.

The rule for lambda expressions Fun is key to the HML system. Like Hindley-Milner, it restricts the type of the parameter to monomorphic types $\tau$ only. As we saw in the introduction, this is essential to avoid guessing polymorphic types. In particular, the invariant that all bounds in $Q$ are non-trivially quantified types $\hat{\varphi}$, ensures that we can never use the parameter $x$ in a polymorphic way.

In regular Implicit MLF, there is no such restriction on $Q$, and one could 'hide' a polymorphic type in the prefix, like $\forall(\alpha \geqslant \mathit{List}\,(\forall\alpha.\,\alpha \to \alpha))$, and assign $\alpha$ to the type of $x$. Since the type $\alpha$ is equivalent to the type $\mathit{List}\,(\forall\alpha.\,\alpha \to \alpha)$ under such prefix, this would allow us to derive a polymorphic type for $x$. Restricting the bounds in $Q$ prevents this from happening.

Note that it is still possible to derive a type for a parameter that has free type variables with a flexible bound. For example, for the abstraction $\lambda x.\,\mathit{choose}\ \mathit{id}\ x$, we can derive the principal type $\forall(\beta \geqslant \forall\alpha.\,\alpha \to \alpha).\,\beta \to \beta$ without further annotations.

The restriction of $Q$ to polymorphic bounds, and the restriction of parameters types to monotypes are the only difference with the type rules for Implicit MLF (and therefore, it follows that the rules for HML are sound). Surprisingly, this restriction is enough to make inference for HML decidable with principal derivations. In contrast to (X)MLF, no rigidly quantified types are needed which simplifies the structure of types, and the resulting type inference algorithm.

In order to pass polymorphic parameters, the rule Fun-Ann must be used where the annotation is restricted to regular System F types $\sigma$. In this rule, the annotated (polymorphic) type of the parameter is assumed in the environment. A nice property of HML is that System F annotations on expressions $e :: \sigma$ can be encoded as an application to an annotated identity function $(\lambda(x :: \sigma).\,x)\ e$. General annotations on expressions would include flexible types though, and in practice we can add the following rule for annotations:

$$\textsc{Ann}\quad \frac{Q,\Gamma \vdash e : \varphi}{Q,\Gamma \vdash (e :: \varphi) : \varphi}$$

### 4.1 Some example derivations

As an example of a complete derivation using these type rules, we derive the type for the expression $\mathit{single}\ \mathit{id}$ under some initial prefix $Q_0$ and environment $\Gamma$. First, we show how we can directly derive the type $\mathit{List}\,(\forall\alpha.\,\alpha \to \alpha)$ for $\mathit{single}\ \mathit{id}$, using $\sigma_{id}$ as a shorthand for $\forall\alpha.\,\alpha \to \alpha$:

$$\frac{\dfrac{\begin{array}{c}Q_0,\Gamma \vdash \mathit{single} : \forall\alpha.\,\alpha \to \mathit{List}\,\alpha \\ Q_0 \vdash \forall\alpha.\,\alpha \to \mathit{List}\,\alpha \sqsubseteq (\sigma_{id} \to \mathit{List}\,\sigma_{id})\end{array}}{Q_0,\Gamma \vdash \mathit{single} : \sigma_{id} \to \mathit{List}\,\sigma_{id}} \qquad Q_0,\Gamma \vdash \mathit{id} : \sigma_{id}}{Q_0,\Gamma \vdash \mathit{single}\ \mathit{id} : \mathit{List}\,(\forall\alpha.\,\alpha \to \alpha)}$$

Note that the instance relation includes the normal System F instance relation and we can immediately instantiate the type of $\mathit{single}$ to $(\forall\alpha.\,\alpha \to \alpha \to \mathit{List}\,(\forall\alpha.\,\alpha \to \alpha)$. Of course, as shown in the introduction, the most general type in HML for $\mathit{single}\ \mathit{id}$ is $\forall(\beta \geqslant \forall\alpha.\,\alpha \to \alpha).\,\mathit{List}\,\beta$, and we can derive this type as follows, using $Q_1$ as a shorthand for $(Q_0, \beta \geqslant \forall\alpha.\,\alpha \to \alpha)$:

$$\frac{\dfrac{\dfrac{\begin{array}{c}Q_1,\Gamma \vdash \mathit{single} : \forall\alpha.\,\alpha \to \mathit{List}\,\alpha \\ Q_1 \vdash \forall\alpha.\,\alpha \to \mathit{List}\,\alpha \sqsubseteq \beta \to \mathit{List}\,\beta\end{array}}{Q_1,\Gamma \vdash \mathit{single} : \beta \to \mathit{List}\,\beta} \qquad \dfrac{\begin{array}{c}Q_1,\Gamma \vdash \mathit{id} : \forall\alpha.\,\alpha \to \alpha \\ Q_1 \vdash \forall\alpha.\,\alpha \to \alpha \sqsubseteq \beta\end{array}}{Q_1,\Gamma \vdash \mathit{id} : \beta}}{Q_1,\Gamma \vdash \mathit{single}\ \mathit{id} : \mathit{List}\,\beta \quad \beta \notin \mathit{ftv}(\Gamma)}}{Q_0,\Gamma \vdash \mathit{single}\ \mathit{id} : \forall(\beta \geqslant \forall\alpha.\,\alpha \to \alpha).\,\mathit{List}\,\beta}$$

Since the prefix contains the assumption $(\beta \geqslant \forall\alpha.\,\alpha \to \alpha)$, we know that $\beta$ will be instance of $\forall\alpha.\,\alpha \to \alpha$ and therefore, we can safely instantiate the type of $\mathit{id}$ to $\beta$. It remains to make this intuition precise, we define the type instance $(\sqsubseteq)$ and type equivalence relation $(\equiv)$ formally in the next section.

## 5. Type equivalence and type instance

Before definining the type instance $(\sqsubseteq)$ and equivalence relation $(\equiv)$, we first we establish some notation for substitutions.

### 5.1 Substitution

A substitution $\theta$ is a function that maps type variables to types. The empty substitution is the identity function and written as $[\,]$. We write $\theta x$ for the application of a substitution $\theta$ 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 $\alpha_i$ to $\sigma_i$ and all other type variables to themselves. The domain of a substitution contains all type variables that map to a different type: $\mathit{dom}(\theta) = \{\alpha \mid \theta\alpha \neq \alpha\}$. The codomain is a set of types and defined as: $\mathit{codom}(\theta) = \{\theta\alpha \mid \alpha \in \mathit{dom}(\theta)\}$. We write $(\alpha := \sigma) \in \theta$ if $\alpha \in \mathit{dom}(\theta)$ and $\theta\alpha = \sigma$. The expression $(\theta - \overline{\alpha})$ removes $\overline{\alpha}$ from the domain of $\theta$, i.e. $(\theta - \overline{\alpha}) = [\alpha := \sigma \mid (\alpha := \sigma) \in \theta \wedge \alpha \notin \overline{\alpha}]$. Finally, we only consider *idempotent* substitutions $\theta$ where $\theta(\theta x) = \theta x$ (and therefore $\mathit{ftv}(\mathit{codom}(\theta)) \mathbin{\#} \mathit{dom}(\theta)$).

### 5.2 A semantic definition of type equivalence and instance

The equivalence relation defines when types are considered equal and abstracts from syntactical artifacts like unbound quantifiers or the order of quantifiers. The instance relation is an extension of the System F instance relation that takes flexible bounds into account. The standard System F instance relation $(\sqsubseteq_{\mathsf{F}})$ is defined as:

$$\frac{\overline{\beta} \mathbin{\#} \mathit{ftv}(\forall\overline{\alpha}.\,\sigma_1)}{\forall\overline{\alpha}.\,\sigma_1 \sqsubseteq_{\mathsf{F}} \forall\overline{\beta}.\,[\overline{\alpha} := \overline{\sigma}]\sigma_1}$$

where we write $(\#)$ for disjoint sets. Note that only outer quantifiers can be instantiated, for example:

$$\begin{array}{lll}\forall\alpha.\,\alpha \to \alpha & \sqsubseteq_{\mathsf{F}} & \mathit{Int} \to \mathit{Int} \\ \forall\alpha.\,\alpha \to \alpha & \sqsubseteq_{\mathsf{F}} & \forall\beta.\,\mathit{List}\,(\forall\alpha.\,\alpha \to \beta) \to \mathit{List}\,(\forall\alpha.\,\alpha \to \beta)\end{array}$$

To extend this definition to flexible types, we are going to interpret flexible types as *sets of System F types*. We can naturally interpret a type $\forall(\alpha \geqslant \sigma_1).\,\sigma_2$ as all instances of type $\sigma_2$ where $\alpha$ is an instance of $\sigma_1$. We write $[\![\varphi]\!]$ for the semantics of $\varphi$ as the set of System F types that are instances of $\varphi$. For example, $[\![\forall(\beta \geqslant \forall\alpha.\,\alpha \to \alpha).\,\mathit{List}\,\beta]\!]$ is the set of types:

$$\{\mathit{List}\,(\forall\alpha.\,\alpha \to \alpha), \forall\beta.\,\mathit{List}\,(\beta \to \beta), \mathit{List}\,(\mathit{Int} \to \mathit{Int}), ...\}$$

Following the approach in (Le Botlan and Rémy 2007b) we define the semantics of types formally as:

**Definition 1** (*Semantics of types*): The semantics of a type $\varphi$, written as $[\![\varphi]\!]$ is defined as:

$$\begin{array}{ll}[\![\bot]\!] & = [\![\forall\alpha.\,\alpha]\!] \\ [\![\sigma]\!] & = \{\sigma' \mid \sigma \sqsubseteq_{\mathsf{F}} \sigma'\} \\ [\![\forall(\alpha \geqslant \hat{\varphi}_1).\,\varphi_2]\!] & = \{\sigma' \mid \sigma_1 \in [\![\hat{\varphi}_1]\!], \sigma_2 \in [\![\varphi_2]\!],\end{array}$$

$$\overline{\beta} \# \mathit{ftv}(\forall(\alpha \geqslant \hat{\varphi}_1).\, \varphi_2),$$
$$\sigma' \in [\![\forall\overline{\beta}.\, [\alpha := \sigma_1]\sigma_2]\!]\}$$

The semantics of $\bot$ is equivalent to $\forall\alpha.\,\alpha$. The semantics of an F type $\sigma$ is the instance closure of all its instances in System F. Finally, the semantics of a quantified type scheme $\forall(\alpha \geqslant \hat{\varphi}_1).\,\varphi_2)$ is the union of the semantics of all possible System F instantiations of $\forall\overline{\beta}.\,[\alpha := \sigma_1]\sigma_2$ where $\sigma_1 \in [\![\varphi_1]\!]$, $\sigma_2 \in \varphi_2$ and $\overline{\beta} \# \mathit{ftv}(\forall(\alpha \geqslant \hat{\varphi}_1).\,\varphi_2)$.

Types are considered equivalent whenever their semantics are equal:

**Definition 2** (*Type equivalence*): We write $\varphi_1 \equiv \varphi_2$ for equivalence between types. It holds if and only if $[\![\varphi_1]\!] = [\![\varphi_2]\!]$.

In contrast to (Implicit) MLF we do not have to define equivalence under a prefix since we have restricted bounds to quantified types only. Besides simplifying the definition of equivalence, this also makes it easier to define a type directed System F translation as we'll see later.

The definition of the instance relation must still be stated under a prefix $Q$ though. In particular, we would like to have the following hypothesis rule to hold:

$$\text{I-HYP}\quad \frac{(\alpha \geqslant \hat{\varphi}) \in Q}{Q \vdash \hat{\varphi} \sqsubseteq \alpha}$$

This rule states that whenever we have the assumption that $\alpha$ is an instance of $\hat{\varphi}$ in the prefix $Q$, we can instantiate that type $\hat{\varphi}$ to $\alpha$. The example type derivation of $\mathit{single}\ \mathit{id}$ in the previous section shows an example of the use of this rule.

Therefore, to define instantiation, we first give an interpretation of prefixes. A prefix $Q$ is meant to capture all the possible types that may be substituted for the type variables in the domain of the prefix, and we interpret a prefix as a *set of substitutions*.

**Definition 3** (*Semantics of prefixes*): The semantics of a prefix $Q$ is written as $[\![Q]\!]$, and represents the set of System F substitutions that capture all possible types that can be substituted for the type variables in the domain of $Q$. We define this formally as:

$$\begin{array}{ll} [\![\varnothing]\!] & = \{[\,]\} \\ [\![Q, \alpha \geqslant \hat{\varphi}]\!] & = \{\theta \circ [\alpha := \sigma] \mid \theta \in [\![Q]\!], \sigma \in [\![\hat{\varphi}]\!]\} \end{array}$$

We can now define the instance relation simply as a subset relation on the set of System F instances:

**Definition 4** (*Type instance*): We write $Q \vdash \varphi_1 \sqsubseteq \varphi_2$, if a type $\varphi_2$ is an instance of a type $\varphi_1$ under a prefix $Q$. It holds if and only if forall $\theta \in [\![Q]\!]$, we have $\theta[\![\varphi_2]\!] \subseteq \theta[\![\varphi_1]\!]$.

Usually, we write $\varphi_1 \sqsubseteq \varphi_2$ when $\varnothing \vdash \varphi_1 \sqsubseteq \varphi_2$.

## 5.3 A syntactic definition of equivalence and instantiation

In practice, a semantic definition is not always easy to work with, and we discuss an equivalent formulation of equivalence and instantiation based on syntactic rules that are inductive.

In particular, we can define equivalence as the smallest transitive, symmetric, and reflexive relation that satisfies the rules of Figure 3. All the rules are unsurprising: the rules EQ-FREE, EQ-RENAME, and EQ-COMM ensure that unbound quantifiers are irrelevant, that we can do $\alpha$-renaming, and that quantifiers can be rearranged. The rules EQ-CONTEXT and EQ-PREFIX allows us to apply equivalence aribrarily deep inside a type. Note that EQ-CONTEXT is safe since it is never possible to make a (non-trivial) quantified type $\hat{\varphi}$ equal to an unquantified type.

Similarly, we defined the instance relation as the smallest transitive relation that satisfies the rules of Figure 4. The rule I-EQUIV includes the equivalence relation. The rule I-BOTTOM shows that bottom can be instantiated to any type. The substitution rule I-SUBST allows us to inline an instantiated bound. This rule is essential since a $\sigma$ type can no longer occur as a flexible bound (which must be quantified type $\hat{\varphi}$). The I-CONTEXT and I-PREFIX rules ensure that we can apply instantiation arbitrarily deep inside bounds. Note that the rule I-CONTEXT only applies to instantiations that lead to a quantified type $\hat{\varphi}_2$ (in other cases rule I-SUBST should be used).

$$\text{EQ-VAR}\quad \forall(\alpha \geqslant \hat{\varphi}).\,\alpha \equiv \hat{\varphi}$$

$$\text{EQ-FREE}\quad \frac{\alpha \notin \mathit{ftv}(\varphi_2)}{\forall(\alpha \geqslant \hat{\varphi}_1).\,\varphi_2 \equiv \varphi_2}$$

$$\text{EQ-RENAME}\quad \frac{\beta \notin \mathit{ftv}(\varphi_2)}{\forall(\alpha \geqslant \hat{\varphi}_1).\,\varphi_2 \equiv \forall(\beta \geqslant \hat{\varphi}_1).\,[\alpha := \beta]\varphi_2}$$

$$\text{EQ-CONTEXT}\quad \frac{\hat{\varphi}_1 \equiv \hat{\varphi}_2}{\forall(\alpha \geqslant \hat{\varphi}_1).\,\varphi \equiv \forall(\alpha \geqslant \hat{\varphi}_2).\,\varphi}$$

$$\text{EQ-PREFIX}\quad \frac{\varphi_1 \equiv \varphi_2}{\forall(\alpha \geqslant \hat{\varphi}).\,\varphi_1 \equiv \forall(\alpha \geqslant \hat{\varphi}).\,\varphi_2}$$

$$\text{EQ-COMM}\quad \frac{\alpha_1 \neq \alpha_2 \quad \alpha_1 \notin \mathit{ftv}(\hat{\varphi}_2) \quad \alpha_2 \notin \mathit{ftv}(\hat{\varphi}_1)}{\forall(\alpha_1 \geqslant \hat{\varphi}_1)(\alpha_2 \geqslant \hat{\varphi}_2).\,\varphi \equiv \forall(\alpha_2 \geqslant \hat{\varphi}_2)(\alpha_1 \geqslant \hat{\varphi}_1).\,\varphi}$$

**Figure 3.** Syntactic type equivalence.

$$\text{I-BOTTOM}\quad Q \vdash \bot \sqsubseteq \varphi$$

$$\text{I-EQUIV}\quad \frac{Q \vdash \varphi_1 \equiv \varphi_2}{Q \vdash \varphi_1 \sqsubseteq \varphi_2}$$

$$\text{I-HYP}\quad \frac{(\alpha \geqslant \hat{\varphi}) \in Q}{Q \vdash \hat{\varphi} \sqsubseteq \alpha}$$

$$\text{I-SUBST}\quad \frac{Q \vdash \hat{\varphi} \sqsubseteq \sigma}{Q \vdash \forall(\alpha \geqslant \hat{\varphi}).\,\varphi \sqsubseteq [\alpha := \sigma]\varphi}$$

$$\text{I-CONTEXT}\quad \frac{Q \vdash \hat{\varphi}_1 \sqsubseteq \hat{\varphi}_2}{Q \vdash \forall(\alpha \geqslant \hat{\varphi}_1).\,\varphi \sqsubseteq \forall(\alpha \geqslant \hat{\varphi}_2).\,\varphi}$$

$$\text{I-PREFIX}\quad \frac{(Q, \alpha \geqslant \hat{\varphi}) \vdash \varphi_1 \sqsubseteq \varphi_2}{Q \vdash \forall(\alpha \geqslant \hat{\varphi}).\,\varphi_1 \sqsubseteq \forall(\alpha \geqslant \hat{\varphi}).\,\varphi_2}$$

**Figure 4.** Syntactic type instance.

**Theorem 5** (*Syntactic type equivalence and instance is sound*): The semantic definitions of type equivalence and type instance satisfy all the rules in Figure 3 and Figure 4.

**Conjecture 6** (*Syntactic type equivalence and instance is complete*): Any type equivalence and instance relation can be derived using the rules in Figure 3 and Figure 4.

The proof of soundness is straightforward induction on the equivalence and instance rules and similar to the proof of soundness for the IMLF instance relation in (Le Botlan and Rémy 2007b). In the same work, Rémy and Le Botlan conjecture the completeness of the IMLF relation based on a proof sketch based on an intermediate graphic representation of types (Rémy and Yakobowski 2007).

$$
\begin{array}{lll}
nf(\sigma) & = \sigma & \\
nf(\bot) & = \bot & \\
nf(\forall(\alpha \geqslant \hat{\varphi}_1).\, \varphi_2) & = nf(\varphi_2) & \text{iff } \alpha \notin ftv(\varphi_2) \\
nf(\forall(\alpha \geqslant \hat{\varphi}_1).\, \varphi_2) & = nf(\hat{\varphi}_1) & \text{iff } nf(\varphi_2) = \alpha \\
nf(\forall(\alpha \geqslant \hat{\varphi}_1).\, \varphi_2) & = \forall(\alpha \geqslant nf(\hat{\varphi}_1)).\, nf(\varphi_2) &
\end{array}
$$

**Figure 5.** Normal form.

$$
\begin{array}{ll}
\mathcal{F}[\![\mathsf{x}]\!] & = x \\
\mathcal{F}[\![\Lambda\alpha \cdot \mathsf{e}]\!] & = \mathcal{F}[\![\mathsf{e}]\!] \\
\mathcal{F}[\![\mathsf{e}\ \sigma]\!] & = \mathcal{F}[\![\mathsf{e}]\!] \\
\mathcal{F}[\![\mathsf{e}_1\ \mathsf{e}_2]\!] & = \mathcal{F}[\![\mathsf{e}_1]\!]\ \mathcal{F}[\![\mathsf{e}_2]\!] \\
\mathcal{F}[\![\lambda(\mathsf{x} : \tau) \cdot \mathsf{e}]\!] & = \lambda x.\, \mathcal{F}[\![\mathsf{e}]\!] \\
\mathcal{F}[\![\lambda(\mathsf{x} : \sigma) \cdot \mathsf{e}]\!] & = \lambda(x :: \sigma).\, \mathcal{F}[\![\mathsf{e}]\!]
\end{array}
$$

**Figure 6.** Embedding of System F.

### 5.4 Normal form

For type inference, it is often useful to bring types into *normal form* which makes it easier to compare them. Figure 5 defines the function $nf(\varphi)$ that returns the normal form of a type $\varphi$.

The first two cases state that F-types and $\bot$ are already in normal form. The next two cases deal with trivial bounds: unbound quantifiers are discarded, and variable types are inlined (EQ-VAR). We have the following useful properties for normal forms:

**Properties 7**
**i.** $nf(\varphi) \equiv \varphi$.
**ii.** $nf(\varphi_1) \approx nf(\varphi_2)$ if and only if $\varphi_1 \equiv \varphi_2$.

where we use $\approx$ for equal types up to the order of the quantifiers (Le Botlan 2004).

## 6. Embedding of System F

It is straightforward to translate any System F program into a well-typed HML program since we only require annotations on polymorphic function parameters. Figure 6 defines the function $\mathcal{F}[\![\mathsf{e}]\!]$ that translates a System F term $\mathsf{e}$ to a well-typed HML term. Basically, all type abstractions and applications are removed, and only annotations on lambda bindings with polymorphic types are retained. Writing $\Gamma \vdash_{\mathsf{F}} \mathsf{e} : \sigma$ for the standard System F type system, we have the following theorem:

**Theorem 8** (*Embedding of System F*): If $\Gamma \vdash_{\mathsf{F}} \mathsf{e} : \sigma$ and $ftv(\Gamma) \subseteq Q$, we have $Q, \Gamma \vdash \mathcal{F}[\![\mathsf{e}]\!] : \varphi$ with $Q \vdash \varphi \sqsubseteq \sigma$.

## 7. Alternative designs

In this section we discuss two restrictions of HML that can be interesting in practice.

### 7.1 Restricting parameters

As shown in Section 4, the principal type of the expression $\lambda x.\, choose\ id\ x$ is $\forall(\beta \geqslant \forall\alpha.\alpha \rightarrow \alpha).\, \beta \rightarrow \beta$. This type can be instantiated to either the Hindley-Milner type $\forall\alpha.\, (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$, or the System F type $(\forall\alpha.\, \alpha \rightarrow \alpha) \rightarrow (\forall\alpha.\, \alpha \rightarrow \alpha)$. The latter type is somewhat surprising – are we able to infer a polymorphic type for the parameter after all? Of course, the derivation is sound since we are unable to *use* the parameter polymorphically in the body of the lambda expression. In that sense, the instantiation is equivalent to any other impredicative instantiation.

$$
\begin{array}{lll}
ftype(\varphi) & = ft(nf(\varphi)) & \\
& \text{where} & \\
& \quad ft(\sigma) & = \sigma \\
& \quad ft(\bot) & = \forall\alpha.\, \alpha \\
& \quad ft(\forall(\alpha \geqslant \bot).\, \varphi) & = \forall\alpha.\, ft(\varphi) \\
& \quad ft(\forall(\alpha \geqslant \forall Q.\, \rho).\, \varphi) & = ft(\forall Q.\, [\alpha := \rho]\varphi)
\end{array}
$$

**Figure 7.** Force a flexible type to a System F type.

$$
\textsc{Let-F}\quad \frac{\begin{array}{c} Q, \Gamma \vdash e_1 : \varphi_1 \\ \forall\varphi_0.\ \text{if } Q, \Gamma \vdash e_1 : \varphi_0 \text{ then } Q \vdash \varphi_1 \sqsubseteq \varphi_0 \\ Q, (\Gamma, x : ftype(\varphi_1)) \vdash e_2 : \varphi_2 \end{array}}{Q, \Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \varphi_2}
$$

**Figure 8.** Environment restricted to System F types.

Nevertheless, one may want to disallow derivations where the free type variables in a parameter type have a polymorphic bound. The only change to the type rules would be in the FUN rule:

$$
\textsc{Fun-Mono}\quad \frac{Q, (\Gamma, x : \tau) \vdash e : \sigma \quad \forall Q.\, \tau \equiv \forall\overline{\alpha}.\, \tau}{Q, \Gamma \vdash \lambda x.\, e : \tau \rightarrow \sigma}
$$

With this rule, the principal type of $\lambda x.\, choose\ id\ x$ becomes the Hindley-Milner type $\forall\alpha.\, (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$. Personally, we feel that this restriction is not very useful in practice though but may be considered if one wants to hide flexible types from programmers.

### 7.2 Restricting let-bindings to System F types

As remarked in the introduction we can also restrict HML let-bindings to System F types only. We call this system Rigid HML. The advantage of such restriction is that programmers are no longer exposed to flexible types and only need to give System F type annotations. A drawback is of course that certain let-bindings now require annotations.

In particular, HML will by default instantiate a let-binding to its most general System F type that has the least inner polymorphism, which ensures compatibility with standard Hindley-Milner. A nice property of Rigid HML, is that we can give a clear (but conservative) annotation rule for let-bindings now: *Only let bindings with a higher-rank type may require an annotation.* As a consequence, Rigid HML is a conservative extension of Hindley-Milner. For example, the following expression:

$$\textbf{let } ids = single\ id \textbf{ in } append\ (single\ inc)\ ids$$

is accepted without annotations where $ids$ gets the expected Hindley-Milner type $\forall\alpha.\, List\ (\alpha \rightarrow \alpha)$. To construct a list of polymorphic elements though, we need to add an annotation:

$$\textbf{let } ids = (single\ id :: List\,(\forall\alpha.\, \alpha \rightarrow \alpha)) \textbf{ in } map\ poly\ ids$$

to accept this program in Rigid HML. The annotation rule is clear, but a bit conservative. For example, no annotation is needed when the most general type of a let-binding is a plain System F type, even if it is impredicative or higher-rank. In particular, the following bindings are accepted without further annotations:

$$
\begin{array}{ll}
ids = (single\ id :: List\,(\forall\alpha.\, \alpha \rightarrow \alpha)) & \\
x = id\ ids & (\text{inferred } List\,(\forall\alpha.\, \alpha \rightarrow \alpha)) \\
y = cons\ id\ ids & (\text{inferred } List\,(\forall\alpha.\, \alpha \rightarrow \alpha)) \\
z = append\ ids & (\text{inf. } List\,(\forall\alpha.\, \alpha \rightarrow \alpha) \rightarrow List\,(\forall\alpha.\, \alpha \rightarrow \alpha))
\end{array}
$$

All of the above bindings have a principal type that is a System F type and there is never any ambiguity and therefore no annotations are needed. Only when the most-general type of a let-binding is a flexible type, and we have a *choice* which System F types we can assign, an annotation may be required. By default, Rigid HML always disambiguates flexible types in a predefined way that leads the System F instance with the least inner polymorphism.

It does this using the function $\mathit{ftype}(\cdot)$ defined in Figure 7 which instantiates a flexible type $\varphi$ to an F type $\sigma$ resulting in the least inner polymorphism. For example, for the binding **let** $\mathit{ids} = \mathit{single}\ \mathit{id}$ it will instantiate the type $\forall(\beta \geqslant \forall\alpha.\, \alpha \to \alpha).\, \mathit{List}\, \beta$ to the expected Hindley-Milner type $\forall\alpha.\, \mathit{List}\, (\alpha \to \alpha)$. In the $\mathit{ftype}(\cdot)$ function, the last case in particular implements this strategy where a type of the form $\forall(\alpha \geqslant \forall Q.\, \rho).\, \varphi$ is instantiated to $\forall Q.\, [\alpha := \rho]\varphi$ [1].

Note that restricting let-bindings to System F types makes Rigid HML less expressive than HML, since we can no longer share an expression like $\mathit{ids}$ at different polymorphic instances, i.e. in contrast to HML, the program:

**let** $\mathit{ids} = \mathit{single}\ \mathit{id}$
**in** $(\mathit{map}\ \mathit{poly}\ \mathit{ids}, \mathit{append}\ (\mathit{single}\ \mathit{inc})\ \mathit{ids})$

is always rejected in Rigid HML as there exists no System F type for $\mathit{ids}$ to make this well typed.

Formally, Rigid HML consists of the normal HML type rules given in Figure 2, where the Let rule is replaced by the rule Let-F defined in Figure 8. The rule Let-F requires that the type $\varphi_1$ derived for $e_1$ is the most general type possible, i.e. forall $\varphi_0$ such that $Q, \Gamma \vdash e_1 : \varphi_0$ holds, we have that $Q \vdash \varphi_1 \sqsubseteq \varphi_0$. The restriction to most general types is needed to ensure that we still have principle type derivations. Finally, the derived type for $e_1$ is instantiated to a System F type using the function $\mathit{ftype}(\varphi_1)$ and bound in the environment.

In practice, a language may choose to use rule Let-F if a let binding is unannotated, while still allowing the programmer to annotate such bindings with flexible types, maintaining the expressiveness of original system.

## 8. Unification

Unification of flexible types is more involved than the usual unification used in Hindley-Milner, but is easier than standard MLF unification since there are no rigid bounds, and we no longer have to compute polynomial weights over types. The unification algorithm uses two kinds of substitutions, namely a prefix $Q$ and a normal substitution $\theta$. In actual implementations both substitutions are usually merged and implemented directly using updateable references on type variables (see (Leijen 2008) for a reference implementation). We have as invariants that $Q$ only contains bounds with quantified types, i.e. $(\alpha \geqslant \varphi) \in Q$ implies $\mathit{nf}(\varphi) \neq \rho$ for any $\rho$ (in other words, $\varphi \in \mathcal{Q}$). Dually, the substitution $\theta$ contains only F types: if $\varphi \in \mathit{codom}(\theta)$ then $\mathit{nf}(\varphi) = \sigma$ for some $\sigma$. Before discussing the actual unification algorithm, we first look at two helper functions over prefixes defined in Figure 11.

### 8.1 Split and update

The split function $\mathit{split}(Q, \overline{\alpha})$ splits a prefix in two parts $Q_1$ and $Q_2$ such that $Q_1$ is the only part useful to $\overline{\alpha}$, i.e. $\forall Q.\, \alpha_1 \to \ldots \to \alpha_n \equiv \forall Q_1.\, \alpha_1 \to \ldots \to \alpha_n$ with $\overline{\alpha} = \{\alpha_1, \ldots, \alpha_n\}$. The split function is used for example to generalize over types where the prefix must be split in one part that can be generalized, and another part that cannot. In an actual implementation, this operation can be implemented efficiently using ranked type variables similar to the usual optimization used for generalization over free variables in the environment (Kuan and MacQueen 2007).

The update function $\mathit{update}(Q, \alpha \geqslant \varphi)$ updates a prefix $Q$ that contains a binding for $\alpha$ with a new binding $\varphi$ taking care to maintain the invariants on $Q$. First the prefix is split according to the free type variables in $\varphi$ to ensure that these are properly scoped. If the binding is updated with an unquantified type, we extend the substitution with $[\alpha := \rho]$, otherwise we update the prefix directly with $\alpha \geqslant \varphi$. Similarly, the expression $\mathit{update}(Q, \alpha := \sigma)$ assigns the type $\sigma$ to $\alpha$, and returns a substitution $[\alpha := \sigma]$, updating the prefix $Q$ accordingly. In an actual implementation, the $\mathit{update}$ operation usually just updates a reference directly, and both $\mathit{split}$ and $\mathit{update}$ are more or less artifacts of using explicit substitutions.

```
$\mathit{unify} :: (Q, \sigma_1, \sigma_2) \to (Q, \theta)$
  where $\sigma_1$ and $\sigma_2$ are in normal form
  and $(\alpha \geqslant \varphi) \in Q \;\Rightarrow\; \mathit{nf}(\varphi) \neq \rho$
  and $(\alpha := \varphi) \in \theta \;\Rightarrow\; \mathit{nf}(\varphi) = \sigma$

$\mathit{unify}(Q, \alpha, \alpha) =$
  return $(Q,\ [\,])$

$\mathit{unify}(Q_1, c\ \sigma_1 \ldots \sigma_n, c\ \sigma'_1 \ldots \sigma'_n) =$
  let $\theta_1 \quad = [\,]$
  let $\theta_{i+1} = \theta'_i \circ \theta_i$
  let $(Q_{i+1}, \theta'_i) = \mathit{unify}(Q_i, \theta_i\sigma_i, \theta_i\sigma'_i)$
  return $(Q_{n+1},\ \theta_{n+1})$

$\mathit{unify}(Q, \alpha, \sigma)$ or
$\mathit{unify}(Q, \sigma, \alpha)$ with $(\alpha \geqslant \hat{\varphi}) \in Q \wedge \sigma \notin \mathcal{V}$
  fail if $(\alpha \in \mathit{dom}(Q/\sigma))$    ('occurs' check)
  let $(Q_1, \theta_1) = \mathit{subsume}(Q, \sigma, \hat{\varphi})$
  let $(Q_2, \theta_2) = \mathit{update}(Q_1, \alpha := \theta_1\sigma)$
  return $(Q_2,\ \theta_2 \circ \theta_1)$

$\mathit{unify}(Q, \alpha_1, \alpha_2)$ with $(\alpha_1 \geqslant \hat{\varphi}_1) \in Q \wedge (\alpha_2 \geqslant \hat{\varphi}_2) \in Q$
  fail if $(\alpha_1 \in \mathit{dom}(Q/\hat{\varphi}_2) \vee \alpha_2 \in \mathit{dom}(Q/\hat{\varphi}_1))$
  let $(Q_1, \theta_1, \varphi) = \mathit{unifyScheme}(Q, \hat{\varphi}_1, \hat{\varphi}_2)$
  let $(Q_2, \theta_2) \quad = \mathit{update}(Q_1, \alpha_1 := \alpha_2)$
  let $(Q_3, \theta_3) \quad = \mathit{update}(Q_2, \alpha_2 \geqslant \varphi)$
  return $(Q_3,\ \theta_3 \circ \theta_2 \circ \theta_1)$

$\mathit{unify}(Q, \forall\alpha.\, \sigma_1, \forall\beta.\, \sigma_2) =$
  assume $c$ is a fresh (skolem) constant
  let $(Q_1, \theta_1) = \mathit{unify}(Q, [\alpha := c]\sigma_1, [\beta := c]\sigma_2)$
  fail if $(c \in (\mathit{con}(\theta_1) \cup \mathit{con}(Q_1)))$
  return $(Q_1,\ \theta_1)$
```

**Figure 9.** Unification.

### 8.2 Unification

The unification algorithm is defined in Figure 9 and Figure 10. The function $\mathit{unify}(Q, \sigma_1, \sigma_2)$ takes a prefix $Q$ and two F types $\sigma_1$ and $\sigma_2$ and returns a new prefix $Q'$ and a substitution $\theta$ such that $Q' \vdash \theta\sigma_1 \equiv \theta\sigma_2$. The cases for equal variables and constructors are standard.

The next two cases deal with the unification of a variable $\alpha$, with a binding $(\alpha \geqslant \varphi) \in Q$, and an F type $\sigma$. First, we ensure that no infinite type is unified with the 'occurrence check' $\alpha \notin \mathit{dom}(Q/\varphi)$, where $\mathit{dom}(Q/\varphi)$ returns the *useful* domain of $Q$ with respect to $\varphi$. In usual ML type inference, this is always equivalent to the free type variables of $\varphi$, but in our case some type variables can be referenced indirectly. For example, $\mathit{dom}((\gamma \geqslant \bot, \beta \geqslant \forall\delta.\, \delta \to \gamma)/(\forall\alpha.\, \alpha \to \beta))$ is $\{\beta, \gamma\}$ even though

[1] Another strategy could be to keep types as polymorphic as possible, tranforming $\forall(\alpha \geqslant \varphi_1).\, \varphi_2$ to $[\alpha := \mathit{ftype}(\varphi_1)]\mathit{ftype}(\varphi_2)$ but we rather stay compatible with the expected Hindley-Milner form.

```
$subsume :: (Q, \sigma, \varphi) \to (Q, \theta)$

$subsume(Q, \forall\overline{\alpha}.\, \rho_1, \forall Q_2.\, \rho_2)$
  assume  $dom(Q) \# dom(Q_2)$ and $\overline{c}$ are fresh constants
  let $(Q_1, \theta_1) \ = unify(QQ_2, [\overline{\alpha} := \overline{c}]\rho_1, \rho_2)$
  let $(Q_2, Q_3) = split(Q_1, dom(Q))$
  let $\theta_2 = \theta_1 - dom(Q_3)$
  fail if $(\overline{c} \in (con(\theta_2) \cup con(Q_2)))$
  return $(Q_2,\ \theta_2)$


$unifyScheme :: (Q, \varphi_1, \varphi_2) \to (Q, \theta, \varphi)$
  where $\varphi_1$ and $\varphi_2$ are in normal form

$unifyScheme(Q, \bot, \varphi)$  or  $unifyScheme(Q, \varphi, \bot)$
  return $(Q,\ [],\ \varphi)$

$unifyScheme(Q, \forall Q_1.\, \rho_1, \forall Q_2.\, \rho_2)$
  assume  the domains of $Q$,$Q_1$, and $Q_2$ are disjoint
  let $(Q_3, \theta_3) \ = unify(QQ_1Q_2, \rho_1, \rho_2)$
  let $(Q_4,\ Q_5) = split(Q_3,\ dom(Q))$
  return $(Q_4,\ \theta_3,\ \forall Q_5.\, \theta_3\rho_1)$
```

**Figure 10.** Subsumption and type scheme unification.

$\gamma \notin ftv(\forall\alpha.\, \alpha \to \beta)$. Formally, we say that $\alpha \in dom(Q/\varphi)$ if and only if $Q = (Q_1, \alpha \geqslant \varphi_1, Q_2)$ and $\alpha \in ftv(\forall Q_2.\, \varphi)$.

After the occurrence check, we try to instantiate the type $\varphi$ to the F type $\sigma$ using the *subsume* function. If that succeeds, we update the prefix with the new substitution $\alpha := \theta_1\sigma$.

The next case deals with two variables $\alpha_1$ and $\alpha_2$ bound to two flexible types $\varphi_1$ and $\varphi_2$. Again, we first do an occurence check for both variables, and if that succeeds, we call *unifyScheme* to find the join of $\varphi_1$ and $\varphi_2$, i.e. the most general type $\varphi$ that is an instance of both $\varphi_1$ and $\varphi_2$, and update the prefix afterwards with $\alpha_1 := \alpha_2$, and $\alpha_2 \geqslant \varphi$.

Finally, we have the case of two quantified types $\forall\alpha.\, \sigma_1$ and $\forall\beta.\, \sigma_2$. In this case we replace both quantifiers with a fresh (skolem) constant $c$ and unify the remaining types. For this to work, we assume that we have ordered the quantifiers of the F types in a canonical order, see (Leijen 2007b,a) for details. Moreover, we need to ensure that the skolem constant does not escape into the environment, and that no free type variables unify with such constant. For example, it would be incorrect to unify $\forall\alpha.\, \alpha \to \beta$ with $\forall\alpha.\, \alpha \to \alpha$. The check $c \notin (con(\theta_1) \cup con(Q_1))$ ensures that this is the case, where the function $con(\cdot)$ returns the type constants in the codomain of the substitution $\theta_1$ or the prefix $Q_1$. In an implementation based on updateable references, this check can be done efficiently by checking if the original types $\forall\alpha.\, \sigma_1$ and $\forall\beta.\, \sigma_2$ do not contain a reference to $c$ after the unification (Leijen 2008).

### 8.3 Subsumption and type scheme unification

The subsumption algorithm $subsume(Q, \sigma, \varphi)$ in Figure 10 takes a prefix $Q$, an F-type $\sigma$ and a flexible type $\varphi$, and returns a new prefix $Q'$ and a substitution $\theta$, such that $Q' \vdash \theta\varphi \sqsubseteq \theta\sigma$. The algorithm is very similar to HMF subsumption (Leijen 2007a). It first skolemizes the $\sigma$ type and instantiates $\varphi$, and unifies the remaining unquantified types. Next, we need to ensure that no skolem constants escape into the environment which is done by checking that the relevant parts of the prefix and substitution do not contain any introduced skolem constant.

Finally, $unifyScheme(Q, \varphi_1, \varphi_2)$ in Figure 10 unifies two flexible types and returns a new prefix $Q'$, a subsitution $\theta$ and a type $\varphi$ such that $\varphi$ is the most general type where $Q' \vdash \theta\varphi_1 \sqsubseteq \varphi$ and $Q' \vdash \theta\varphi_2 \sqsubseteq \varphi$. Note how the *split* function is used here to generalize over the result type $\varphi$.

```
(update the prefix)
$update(Q, \alpha \geqslant \varphi_2)$
  let $(Q_0, (Q_1, \alpha \geqslant \hat{\varphi}_1, Q_2)) = split(Q, ftv(\varphi_2))$
  if $(nf(\varphi_2) = \rho)$
    then return $((Q_0,\ Q_1, [\alpha := \rho]Q_2),\ [\alpha := \rho])$
    else return $((Q_0,\ Q_1, \alpha \geqslant \varphi_2,\ Q_2),\ [\,])$

$update(Q, \alpha := \sigma)$
  let $(Q_0, (Q_1, \alpha \geqslant \hat{\varphi}, Q_2)) = split(Q, ftv(\sigma))$
  return $((Q_0,\ Q_1, [\alpha := \sigma]Q_2),\ [\alpha := \sigma])$


(extend a prefix
$extend(Q, \alpha \geqslant \varphi) =$
  if $(nf(\varphi) = \rho)$
    then return $(Q,\ [\alpha := \rho])$
    else return $((Q, \alpha \geqslant \varphi),\ [\,])$

(split a prefix)
$split :: (Q, \overline{\alpha}) \to (Q,\ Q)$

$split(\varnothing, \overline{\alpha}) =$
  return $(\varnothing,\ \varnothing)$

$split((Q, \alpha \geqslant \hat{\varphi}), \overline{\alpha}) =$  with $\alpha \in \overline{\alpha}$
  let $(Q_1,\ Q_2) = split(Q, ((\overline{\alpha} - \alpha) \cup ftv(\varphi)))$
  return $((Q_1, \alpha \geqslant \hat{\varphi}),\ Q_2)$

$split((Q, \alpha \geqslant \hat{\varphi}), \overline{\alpha}) =$  with $\alpha \notin \overline{\alpha}$
  let $(Q_1,\ Q_2) = split(Q, \overline{\alpha})$
  return $(Q_1,\ (Q_2, \alpha \geqslant \hat{\varphi}))$
```

**Figure 11.** Helper functions.

## 9. Type inference

The type inference algorithm for HML is defined in Figure 12 and is very similar to MLF inference. It is surprisingly straightforward since the unification algorithm is rather powerful. The rule for variables just looks up the type in the environment. Inference for **let** bindings is very simple and we only need to bind the inferred type for the bound expression in the environment.

For a lambda expression, we assume a fresh type $\alpha$ for the parameter $x$. After doing inference for the body, we check if the inferred type for $x$ is a monotype. This is to prevent inferring polymorphic types for function parameters as required by the FUN type rule. This check is sufficient because we also ensure that $Q$ only contains polymorphic bounds. Afterwards *split* is used to generalize over the result type. The result type is flexibly quantified using $(\beta \geqslant \varphi_1)$. For example, the most general type for the *const* function is:

$$\begin{array}{ll} const :: \forall\alpha.\, \forall(\gamma \geqslant \forall\beta.\, \beta \to \alpha).\, \alpha \to \gamma & \text{(inferred)} \\ const = \lambda x.\, \lambda y.\, x & \end{array}$$

where its type can be instantiate to both of the usual System F types that we can assign to the *const* function:

$$\begin{array}{lcl} \forall\alpha.\, \forall(\gamma \geqslant \forall\beta.\, \beta \to \alpha).\, \alpha \to \gamma & \sqsubseteq & \forall\alpha\beta.\, \alpha \to \beta \to \alpha \\ \forall\alpha.\, \forall(\gamma \geqslant \forall\beta.\, \beta \to \alpha).\, \alpha \to \gamma & \sqsubseteq & \forall\alpha.\, \alpha \to (\forall\beta.\, \beta \to \alpha) \end{array}$$

This means that we need no special rules for ‘deep instantiation’ or prenex conversions on types (Peyton Jones et al. 2007).

| System | Types | Flexible types | Annotations |
|---|---|---|---|
| HMF | Regular F-types | No | On polymorphic parameters and ambiguous impredicative applications |
| Rigid HML | Flexible F-types | No | On polymorphic parameters and let bindings with higher-rank types |
| HML | Flexible F-types | Let-bound only | On polymorphic parameters |
| MLF | Flexible and Rigid | Let- and $\lambda$-bound | On parameters that are used polymorphically |

**Figure 13.** A high level comparision between different inference systems.

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

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

$infer(Q, \Gamma,$ let $x = e_1$ in $e_2)$
    let $(Q_1, \theta_1, \varphi_1) = infer(Q, \Gamma, e_1)$
    let $(Q_2, \theta_2, \varphi_2) = infer(Q_1, (\Gamma, x : \varphi_1), e_2)$
    return $(Q_2, \theta_2 \circ \theta_1, \varphi_2)$

$infer(Q, \Gamma, \lambda x. e) =$
    assume $\alpha, \beta$ are fresh
    let $(Q_1, \theta_1, \varphi_1) = infer((Q, \alpha \geqslant \bot), (\Gamma, x : \alpha), e)$
    fail if not $(\theta_1 \alpha = \tau)$ for some $\tau$
    let $(Q_2, Q_3)$     $= split(Q_1, dom(Q))$
    let $(Q'_3, \theta'_3)$     $= extend(Q_3, \beta \geqslant \varphi_1)$
    return $(Q_2, \theta_1, \forall Q'_3. \theta_1 \alpha \rightarrow \theta'_3 \beta)$

$infer(Q, \Gamma, e_1\ e_2) =$
    assume $\alpha_1, \alpha_2, \beta$ are fresh
    let $(Q_1, \varphi_1, \theta_1) = infer(Q, \Gamma, e_1)$
    let $(Q_2, \varphi_2, \theta_2) = infer(Q_1, \Gamma, e_2)$
    let $(Q'_2, \theta'_2)$  $= extend(Q_2, \alpha_1 \geqslant \theta_2 \varphi_1, \alpha_2 \geqslant \varphi_2, \beta \geqslant \bot)$
    let $(Q_3, \theta_3)$  $= unify(Q'_2, \theta'_2 \alpha_1, \theta'_2 \alpha_2 \rightarrow \beta)$
    let $(Q_4, Q_5) = split(Q_3, dom(Q))$
    return $(Q_4, \theta_3 \circ \theta_2 \circ \theta_1, \forall Q_5. \theta_3 \beta)$
```

**Figure 12.** Type inference.

The application case first infers the types $\varphi_1$ and $\varphi_2$ for the expressions $e_1$ and $e_2$. It then unifies the types $\alpha_1$ and $\alpha_2 \rightarrow \beta$ under a prefix that assumes $(\alpha_1 \geqslant \theta_2\varphi_1)$, $(\alpha_2 \geqslant \varphi_2)$, and an unconstrained $(\beta \geqslant \bot)$. Finally, it generalizes the result using *split*.

**Theorem 9** (*Soundness of inference*): If $infer(Q_0, \Gamma, e)$ succeeds with $(Q, \theta, \varphi)$, then $Q, \Gamma \vdash \theta e : \theta\varphi$ holds and $Q_0 \sqsubseteq Q$.

**Theorem 10** (*Completeness of inference*): Assume a $Q_0$ where $ftv(\Gamma) \subseteq dom(Q_0)$ and $Q_0 \sqsubseteq Q$. If $Q, \Gamma \vdash e : \varphi$ holds then $infer(Q_0, \Gamma, e)$ succeeds too with a principal solution $(Q', \theta', \varphi')$, such that $Q' \vdash \theta'\varphi' \sqsubseteq \varphi$ and $Q' \sqsubseteq Q$.

### 9.1 Implementing alternatives

Implementing the FUN-MONO rule of Section 7.1 is straightforward. In the case for lambda expressions, we simply instantiate the parameter type to its usual Hindley-Milner type, and ensure that the result type is of rank 1:

```
$infer(Q, \Gamma, \lambda x. e) =$
    assume $\alpha, \beta$ are fresh
    let $(Q_1, \theta_1, \varphi_1) = infer((Q, \alpha \geqslant \bot), (\Gamma, x : \alpha), e)$
    let $\tau_1 = \theta_1 \alpha$
    let $(Q_2, Q_3) = split(Q_1, dom(Q))$
    let $(Q_4, Q_5) = split(Q_3, ftv(\tau_1))$
    fail if $(ftype(\forall Q_4. \tau_1) \neq \forall \overline{\alpha}. \tau)$ for some $\overline{\alpha}, \tau$
    let $(Q'_5, \theta'_5)$     $= extend(Q_5, \beta \geqslant \varphi_1)$
    return $(Q_2, \theta_1, \forall \overline{\alpha}. \forall Q'_5. \tau \rightarrow \theta'_5 \beta))$
```

We use the *ftype* function (see Figure 7) to instantiate to a type with the least inner polymorphism which is the only instance that can be a rank-1 type. Note how we first split the prefix $Q_3$ with respect to the free variables of $\tau_1$ such that the prefix $Q_5$ stays as polymorphic as possible.

The Rigid HML variant is even more straightforward to implement, since we just need to call the *ftype* function when binding the value in the environment:

```
$infer(Q, \Gamma,$ let $x = e_1$ in $e_2)$
    let $(Q_1, \theta_1, \varphi_1) = infer(Q, \Gamma, e_1)$
    let $(Q_2, \theta_2, \varphi_2) = infer(Q_1, (\Gamma, x : ftype(\varphi_1)), e_2)$
    return $(Q_2, \theta_2 \circ \theta_1, \varphi_2)$
```

and no further changes are needed.

## 10. Related work

MLF was first described by by Rémy and Le Botlan (2004; 2003; 2007a; 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 2007b), 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.

Rémy and Le Botlan introduce Implicit MLF (2007b) as a simpler version of MLF where only let-bindings can have types with flexible quantification. This simplification allows them to give a semantics of flexible types in terms of sets of System F types. Unfortunately, inference for Implicit MLF is undecidable. The variant with type annotations is introduced as XMLF and uses rigid quantification to enable type inference.

Leijen describes HMF (Leijen 2007a): a type inference system for first-class polymorphism that is based on regular Hindley-Milner inference. Annotations are required on function parameters with a polymorphic type and on all ambiguous impredicative applications. Even though the annotation rule is harder than that of HML or MLF, it represents an interesting design point since the specification and the type inference algorithm are very simple: the declarative type rules use only regular System F types and the inference algorithm is a small extension of algorithm W.

As a reference, Figure 13 gives a high-level comparison between these proposed type systems for first-class polymorphism. Note that flexible types add true expressiveness to a type system since it allows the sharing of values at different polymorphic instances (as shown in Section 2.1 and Section 7.2).

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).

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.

## 11. Conclusion

We presented HML, a type inference system that supports full first-class polymorphism with few annotations: only function parameters with a polymorphic type need to be annotated. It has logical type rules with principal derivations where flexible types ensure that every expression can be assigned a most general type. The type inference algorithm is straightforward and we have implemented a reference implementation that represents substitutions using updateable references and uses ranked type variables to do efficient generalization (Leijen 2008).

Given these good properties of HML, we think it can be an excellent type system in practice for languages that support first-class polymorphic values.

## 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.

George Kuan and David MacQueen. Efficient ML type inference using ranked type variables. In *The 2007 ACM SIGPLAN Workshop on ML (ML 2007)*, Freiburg, Germany, October 2007.

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 2007a.

Didier Le Botlan and Didier Rémy. Recasting MLF. Research Report 6228, INRIA, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, June 2007b.

Daan Leijen. HMF: Simple type inference for first-class polymorphism. Technical Report MSR-TR-2007-118, Microsoft Research, September 2007a.

Daan Leijen. A reference implementation of HML. Available at `http://research.microsoft.com/users/daan/pubs.html`, April 2008.

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

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

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

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.

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.

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