# Qualified Types for MLF

Daan Leijen  Andres Löh

Institute of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

Draft: Revision: 166

{daan,andres}@cs.uu.nl

## Abstract

MLF is a type system that extends a functional language with impredicative rank-$n$ polymorphism. Type inference remains possible and only in some clearly defined situations, a local type annotation is required. Qualified types are a general concept that can accommodate a wide range of type systems extension, for example, type classes in Haskell. We show how the theory of qualified types can be used seamlessly with the higher-ranked impredicative polymorphism of MLF, and give a solution to the non-trivial problem of evidence translation in the presence of impredicative datatypes.

## 1. Introduction

MLF [6] is a type system that extends a functional language (in the style of ML [8] or Haskell [9]) with a form of impredicative rank-$n$ polymorphism. Type inference in the extended system remains possible, but in some clearly defined situations, code that makes use of polymorphic arguments must be locally annotated with types.

Applications that require rank-$n$ polymorphism are surprisingly ubiquitous in advanced functional programming. Shan [11] as well as Peyton Jones and Shields [10] present plenty of convincing examples, such as dynamic types, datatype invariants, and generic (polytypic) functions. Current Haskell implementations are therefore already equipped with a type system that supports a limited form of rank-$n$ polymorphism.

The excellent tutorial paper by Peyton Jones and Shields [10] describes the ideas behind the Haskell implementation in great detail and explains many of the design decisions made.

A significant limitation of the Haskell implementation with respect to the MLF system is that the former is predicative: a quantified variable can only range over monomorphic types, whereas in MLF, a type variable can be instantiated to a polymorphic type again. Impredicativity is essential for abstraction over polymorphic values and has a number of advantages for the programmer that make MLF more flexible and more intuitive to use.

However, there are two reasons that currently prevent a wider adoption of the more general MLF approach:

- In MLF, quantifiers have bounds that are all collected in a prefix at the beginning of the type. This requires the reader to perform an on-the-fly substitution, which makes it much harder to understand a complicated type. Also, it just looks plainly unfamiliar.
- Functional programming languages used in practice already possess other extensions to the type system besides rank-$n$ polymorphism. Many of these type system extensions, such as type classes, implicit parameters, or records, are based on the theory of qualified types [3]. Due to impredicativity, it is unclear how MLF can work in conjunction with such extensions.

We have the firm belief that both doubts just raised can be eliminated, and we make the following contributions:

- In Section 2.6 we present a simple convention that can be used to write MLF types in a more intuitive way. The additional notational complexity that bounded quantification introduces is only imposed on the programmer when it plays a crucial role.

- In Section 5 we introduce a variant of the MLF type inference algorithm which performs type-directed evidence translation to System F. This is the key contribution of this paper, because it allows the addition of qualified types to MLF in a way that it can be used efficiently in an actual implementation.

With the addition of qualified types, we believe that this paper is a substantial step forward to making MLF fit for use in practical programming languages.

The paper starts by introducing the two main topics that we want to connect: The MLF type system is explained by example in Section 2, comparing it to a plain Hindley-Milner type system without rank-$n$ polymorphism and the system used in current Haskell implementations. Section 3 introduces qualified types, giving examples of a number of different applications, including the Haskell type class system. We also show that the sum of the two features, rank-$n$ and qualified types, is greater than its parts, by pointing out applications that are only possible with impredicative qualified types.

Interestingly, the difficult part in combining the features is not to extend the MLF type system with qualified types, but how to implement the resulting system. Qualified types are usually implemented using evidence translation, explained in Section 4. We demonstrate why a naïve extension of the standard translation scheme fails, and we present the core ideas to overcome this problem.

In Section 5, we present a variant of the MLF type inference algorithm that is augmented with a type-directed evidence translation to System F. Adding actual qualified types to the system is then easy and explained in Section 6. In Section 7, we conclude and discuss future work.

## 2. A tour of MLF

In this section, we introduce first-class polymorphism and analyze the advantages that MLF offers over Hindley-Milner and other implementations of higher ranked types.

### 2.1 Higher-ranked types

The Hindley-Milner type system [2] stands at the basis of almost all polymorphic programming language type systems. An important property of the Hindley-Milner type system is that there exists a type inference algorithm that automatically infers most general types for expressions, and there is no need for extra type annotations. At the same time, the type system is sound, where the execution of a well-typed program can not "go wrong". The combination of these features, together with a straightforward implementation, makes the Hindley-Milner type system well-suited for practical use.

However, the Hindley-Milner type system has a significant limitation: polymorphic values are not first-class. In practice this means that only values that are bound by a **let** construct can have a polymorphic type. It is not possible for the argument of a function to be used with a polymorphic type. Here is an example of a function that takes a polymorphic argument[1]:

$$f\ choose = (choose\ True\ False, choose\ \texttt{'a'}\ \texttt{'b'})$$

The function $f$ takes a function $choose$ and calls it on two booleans and two characters in the body of the function. Neither Haskell, nor ML, would accept this definition, because in the Hindley-Milner type system, lambda-bound variables such as $choose$ can only have monomorphic types, which is equivalent to the property that universal quantifiers can only appear at the outermost level of a type.

In contrast, explicitly typed languages such as System F allow universal quantifiers to appear deep within a type. For example, if we know that $choose$ has the polymorphic type $\forall\alpha.\,\alpha \to \alpha \to \alpha$, then the above definition makes sense, and $f$ can be given the following type:

$$(\forall\alpha.\,\alpha \to \alpha \to \alpha) \to (Bool, Char)$$

[1] Note that we adopt a Haskell-like notation throughout this paper, even though we are talking about a type system that has been introduced as an extension of ML.

This is a rank-2 type, as it contains quantifiers to the left of the function arrow. The Haskell implementations GHC and Hugs support higher-rank polymorphism such as it occurs in the definition of $f$ above. If $f$ is equipped with a type signature, the compiler accepts the definition. However, even though higher-ranked, the implementations are still limited due to a second restriction of Hindley-Milner: quantified variables can only be instantiated with a monomorphic type. Systems with this restriction are called *predicative*.

The MLF type system is an extension of Hindley-Milner that fully supports first-class polymorphism: universal quantifiers can appear anywhere in a type, and quantified variables can be instantiated with a polymorphic type. Because of the last feature, MLF is an *impredicative* type system.

We believe that impredicativity is very important for abstraction and we discuss its advantages in more detail in Section 2.4. First, we take a closer look at the difference between predicative and impredicative instantiation. Take for example the following two functions:

$$\begin{aligned} &choose :: \forall\alpha.\, \alpha \to \alpha \to \alpha \\ &id \quad\;\; :: \forall\beta.\, \beta \to \beta \end{aligned}$$

Let us consider the application $choose\ id$. In a predicative system, the type of $id$ has to be instantiated to a fresh $\gamma$, because as an argument to $choose$, it cannot have a polymorphic type. Consequently, we get:

$$choose\ id :: \forall\gamma.\, (\gamma \to \gamma) \to (\gamma \to \gamma)$$

This is the type derived by GHC for example. Impredicatively, we have a choice. We can do the same as above, but we can also instantiate $\alpha$ to the polymorphic type $\forall\beta.\, \beta \to \beta$:

$$choose\ id :: (\forall\beta.\, \beta \to \beta) \to (\forall\beta.\, \beta \to \beta)$$

However, the $choose\ id$ example immediately reveals a problem with impredicative polymorphism: neither of the two types above is an instance of the other. Indeed, a naïve implementation of impredicative polymorphism does not have principal types.

### 2.2 Principal types

A type system is said to have principal types if each expression can be assigned a "best possible type", of which all other types it could have are instances. Principal types are necessary for type inference to work efficiently, because it allows that optimal types are inferred by looking only at a part of the program, rather than having to perform a global analysis.

The achievement of MLF is that it restores the principal types property even in the presence of impredicativity. It does so by extending the type language, so that we can assign a type to $choose\ id$ of which both types above are instances. As a consequence, MLF does support efficient type inference. The principal type for $choose\ id$ in MLF is

$$choose\ id :: \forall(\alpha \geq \forall\beta.\, \beta \to \beta).\, \alpha \to \alpha$$

The quantification on $\alpha$ gets a *bound*, meaning that it can be instantiated only with types that are an instance of $\forall\beta.\, \beta \to \beta$. Instantiating $\alpha$ with $\gamma \to \gamma$ for a fresh $\gamma$, or with $\forall\beta.\, \beta \to \beta$, leads to the two types of $choose\ id$ given above.

### 2.3 Type annotations

Type inference in MLF does never invent polymorphism: whenever a lambda-bound argument is used polymorphically, a type annotation is required. The same holds for the Haskell implementation of higher-ranked polymorphism.

All these systems, however, share the property that type annotations are only required for programs which actually make use of higher-ranked polymorphism. All other programs continue to work without type annotations.

Therefore, switching the underlying type system of an ML-like language to MLF will not break any existing programs.

### 2.4 Advantages of MLF

Until now, we have seen that we can handle impredicative types by complicating the type language. We will argue in Section 2.6 that this complication is not very serious. But first, we want to show what we *gain* by having the MLF system rather than, for instance, the Haskell implementation of higher-ranked types.

We will present two significant examples.

***Function application is not a special construct*** Consider the function

$$(\$) :: \forall \alpha\, \beta.\, (\alpha \to \beta) \to \alpha \to \beta$$
$$(\$) = id$$

defined in the Haskell prelude. This function replaces function application, such that we can write $f\ \$\ x$ instead of $f\ \ x$ to denote the application of $f$ to $x$. Fixity rules in Haskell specify that $\$$ is right-associative and binds very weakly, whereas ordinary function application is left-associative and binds very strongly. The operator $(\$)$ is often used to remove the need for parentheses that would otherwise extend over large portions of code.

In the context of higher-ranked types, however, $(\$)$ in Haskell is strictly weaker than normal function application, because the quantified variables $\alpha$ and $\beta$ in the type of $(\$)$ cannot be instantiated to polymorphic types. We show this with the help of well-known higher-ranked function $runST$, which is part of the Haskell libraries:

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

This function was one of the first motivations to consider higher-ranked types in Haskell. The function $runST$ is used to evaluate a stateful computation of type $ST\ \gamma\ \alpha$, yielding a value of type $\alpha$. The computation must be independent of the parameter $\gamma$, which ensures that the state does not escape from the computation and is not used in other stateful computations. Values of type $ST\ \gamma\ \alpha$ are often complex and extend over multiple lines of code, therefore it would be tempting to write

$$runST\ \$\ very\ complicated\ computation$$

This, however, is rejected by all Haskell implementations. However, the expression

$$runST\ (very\ complicated\ computation)$$

is accepted. With the impredicative MLF type system, both versions work without type annotations.

This example stands for a whole class of applications of impredicativity which leads to less type annotations required, or actually more programs becoming admissible. For instance,

$$id\ runST\ (very\ complicated\ computation)$$

is not possible to write in Haskell either for much the same reason ($id$ cannot be instantiated to $runST$'s polymorphic type). In effect, the usual abstraction mechanism of polymorphism no longer works for higher-ranked types in Haskell. This means that the programmer needs to define specific versions of general functions like $id$ and $(\$)$ for each kind of higher-rank type occurring in the program.

The special role of function application is not merely an inconvenience to the programmer, it makes it also more difficult to prove theorems, perform program derivations, or any kind of source-to-source transformations. Many common transformations such as

$$id\ x \equiv x$$

are suddenly subject to side conditions which require looking at the type of arguments that are involved! With MLF, neither programmers nor tool developers need to consider rank-$n$ types as something special, and the usual abstraction mechanism remain valid.

***Data structures over polymorphic types*** In a predicative system, not only function application is special in being able to deal with polymorphic types, also the function space type constructor $(\rightarrow)$ takes a special role: it is the only type constructor that can be parameterized over polymorphic types.

In an impredicative system, any parameterized datatype can be instantiated to polymorphic types just as it can be instantiated to monomorphic types. With MLF, we can build lists, tuples, trees, or complex abstract datatypes such as Haskell's $IO$ monad which are parametrized over a polymorphic type.

Recall the rank-2 polymorphic function $runST$:

$$runST :: \forall\alpha.\,(\forall\gamma.\,ST\ \gamma\ \alpha) \rightarrow \alpha$$

In Haskell, it is not possible to place this function in a list or a pair – both of the expressions

$$[runST]$$
$$(runST, \texttt{'a'})$$

are rejected by the Haskell type checker, and this situation cannot be improved by providing type signatures. The reason is that the types would have to be

$$[\forall\alpha.\,(\forall\gamma.\,ST\ \gamma\ \alpha) \rightarrow \alpha]$$
$$(\forall\alpha.\,(\forall\gamma.\,ST\ \gamma\ \alpha) \rightarrow \alpha, Char)$$

respectively, and these are just not legal in a predicative system. With MLF as type system, both expressions are accepted.

It should be mentioned that Haskell allows to place polymorphic values into data structures by hiding the polymorphism within a datatype. One can define

$$\mathbf{newtype}\ RunSTType = Pack\ (\forall\alpha.\,(\forall\gamma.\,ST\ \gamma\ \alpha) \rightarrow \alpha)$$
$$unPack\ (Pack\ x) = x$$

and then use $Pack\ runST$ to store the value into a data structure, and $unPack$ after we extract it from the data structure.

The disadvantage of this approach is immediately obvious: while $\mathbf{newtype}$ can be implemented efficiently and causes no runtime overhead, it is very tedious to use this technique as a programmer. Separate datatypes to pack polymorphic values are needed for each polymorphic type.

We believe that the current situation, while theoretically no less expressive, in practice prevents the programmer from adopting a solution which makes use of polymorphic types. The MLF type system encourages programmers to see polymorphic values as first-class values in every respect, that need not be avoided and require no special treatment.

### 2.5 MLF formally

After having looked at the features of the MLF system, let us now introduce the MLF type language more formally. The MLF type language distinguishes monomorphic types (*monotypes*) from polymorphic types (*polytypes*). The syntax of monotypes is defined as:

$$\tau ::= g\ \tau_1 \ldots \tau_n \mid \alpha$$

In other words, a monotype is either an applied constructor $g$, or a type variable. We assume that the binary function space constructor $(\rightarrow)$ is among the possible constructors.

A polytype is either bottom $(\bot)$, or a monotype quantified with a (possibly empty) *prefix* $Q$:

$$\sigma ::= \bot \mid \forall Q.\, \tau$$

A Prefix is a list of constraints that associate a type variable with a polytype. Each constraint is of the form:

$$(\alpha \diamond \sigma)$$

Here, $\diamond$ stands either for $=$, in which case the constraint is called a *rigid bound*, or it stands for $\geqslant$, and then we speak of a *flexible bound*.

A bound $(\alpha \diamond \sigma)$ quantifies the type variable $\alpha$ in the rest of the prefix and the qualified type. The idea is that a rigid bound can only be instantiated with *exactly* the type given, whereas a flexible bound can be instantiated with any *instance* of the given type.

We will not describe the instance relation precisely, but will focus on some important properties of instantiation. First of all, the instance relation is transitive and reflexive, where every type is an instance of $\bot$ (written $\bot \sqsubseteq \sigma$), thus if bottom is used in a flexible bound, this represents ordinary unbounded quantification.

For example, the identity function has the MLF type

$$\forall(\alpha \geq \bot).\, \alpha$$

We abbreviate unbounded quantification and write

$$\forall\alpha.\, \alpha$$

instead.

We have seen a nontrivial MLF type for $choose\ id$,

$$\forall(\alpha \geq \forall\beta.\, \beta \to \beta).\, \alpha \to \alpha$$

where $\alpha$ can be instantiated with both $\forall\beta.\, \beta \to \beta$ itself and $\gamma \to \gamma$ for a fresh variable $\gamma$. Formally,

$$\forall(\alpha \geq \forall\beta.\, \beta \to \beta).\, \alpha \to \alpha \sqsubseteq \forall(\alpha = \forall\beta.\, \beta \to \beta).\, \alpha \to \alpha$$
$$\forall(\alpha \geq \forall\beta.\, \beta \to \beta).\, \alpha \to \alpha \sqsubseteq \forall\gamma.\, \gamma \to \gamma \to (\gamma \to \gamma)$$

Bounded quantification allows us to express flexibility while at the same time maintaining a relation between different parts of the type that are represented by the same variable.

Rigid bounds occur whenever a function requires a polymorphic argument, such as $runST$, which has MLF type

$$\forall\alpha\,(\beta = \forall\gamma.\, ST\ \gamma\ \alpha).\, \beta \to \alpha$$

Here, we can not instantiate the type as a truly polymorphic type is required. Since this restricts instantiation, we can turn flexible bounds into rigid bounds (but not the other way around), in other words: $\forall(\alpha \geq \sigma_\alpha).\, \sigma \sqsubseteq \forall(\alpha = \sigma_\alpha).\, \sigma$.

Besides the instance relation on types there is also an equivalence relation on polytypes, written $\equiv$. For instance, the type

$$\forall\gamma.\, \gamma \to \gamma \to (\gamma \to \gamma)$$

is equivalent to the more complicated

$$\forall\gamma\,(\beta = \gamma \to \gamma).\, \beta \to \beta$$

Equivalence preserves the free type variables in a type. The equivalence relation can be used to compute a *normal form* of an MLF type. The algorithm to compute the normal form is shown in Figure 1. The labels on the bounds $v$ can be ignored for now – they will only become relevant in Section 5 when these are used for the type directed translation. The normal form algorithm basically simplifies *trivial* bounds away. From the definition we

$$
\begin{array}{lll}
\mathsf{nf}(\tau) & \doteq \tau & \\
\mathsf{nf}(\bot) & \doteq \bot & \\
\mathsf{nf}(\forall(\alpha \diamond_v \sigma_\alpha).\, \sigma) & \doteq \mathsf{nf}(\sigma) & \text{if } \alpha \notin \mathsf{ftv}(\sigma) \\
\mathsf{nf}(\forall(\alpha \diamond_v \sigma_\alpha).\, \sigma) & \doteq \mathsf{nf}(\sigma_\alpha) & \text{if } \mathsf{nf}(\sigma) = \alpha \\
\mathsf{nf}(\forall(\alpha \diamond_v \sigma_\alpha).\, \sigma) & \doteq \mathsf{nf}(\sigma)\ [\alpha \mapsto \tau] & \text{if } \mathsf{nf}(\sigma_\alpha) = \tau \\
\mathsf{nf}(\forall(\alpha \diamond_v \sigma_\alpha).\, \sigma) & \doteq \forall(\alpha \diamond_v \mathsf{nf}(\sigma_\alpha)).\, \mathsf{nf}(\sigma) &
\end{array}
$$

**Figure 1.** Normal form of MLF types

can see that there are three forms of trivial bounds: the quantified variable is bound by a monotype, the variable does not occur in the body of the type, or if the body of the type is itself a variable.

THEOREM 1. *The normal form of a type is always equivalent to the type itself, i.e.,*

$$\mathsf{nf}(\sigma) \equiv \sigma$$

For a more thorough explanation of the MLF theory including the exact definitions of instantiation and equivalence, the reader is referred to the MLF article [6] or thesis [5].

### 2.6 Presentation of MLF types

MLF types put us in a dilemma: they are nice to work with from a theoretician's or an implementor's point of view, but they are awkward to read for a programmer. The reason for both is that MLF types, although higher-ranked, collect all the qualifiers in a prefix, at the beginning of the type.

In some cases the normal form computation can help to simplify the presentation of a type: if a quantified variable is bounded by a monotype, if a variable does not occur in the body of the type, or if the body of the type is itself a variable. But even the normal form can still be unexpectedly verbose. For instance, the type of $runST$ in MLF is

$$\forall\alpha\,(\beta = \forall\gamma.\, ST\ \gamma\ \alpha).\, \beta \to \alpha$$

and this type is in normal form. Yet, the originally given type

$$\forall\alpha.\, (\forall\gamma.\, ST\ \gamma\ \alpha) \to \alpha$$

would probably be considered to be much more readable by most programmers. We therefore propose to adopt a simple heuristic when presenting types to users in MLF-based systems:

- a flexible constraint $(\alpha \geq \sigma)$ is inlined when $\alpha$ appears only in *positive* positions, i.e., to the right of a function arrow,
- a rigid constraint $(\alpha = \sigma)$ is inlined when $\alpha$ appears only in *negative* positions, i.e., to the left of a function arrow,
- other type constructors do not influence the sign of the position.

This heuristic is loss-free: the original MLF type can easily be recovered from the simplified type.

As an example, consider the following explicit MLF type

$$\forall(\beta = \forall\alpha.\, \alpha \to \alpha)\,(\gamma \geq \forall\alpha.\, \alpha \to \alpha).\, \beta \to \gamma$$

can be presented according to this heuristic as:

$$[\forall\alpha.\, \alpha \to \alpha] \to [\forall\alpha.\, \alpha \to \alpha]$$

The heuristic is used in the experimental Morrow compiler [7], which presents the above type and the type of $runST$ in their simpler form. Experience to the present day shows that it makes MLF very easy to work with, because the bounded quantifications that remain are the ones where the expressed sharing actually plays a crucial role in the type. Most simple functions, however, get the same types as they would in ML or Haskell. In the remainder of this article we will make use of this convention.

## 3. Qualified types

Jones introduced the theory of qualified types [3] to generally describe a wide range of type system extensions, ranging from ad-hoc overloading to record operations. The main reason to consider qualified types in conjunction with MLF is that it gives us a general framework to easily extend MLF with features that are essential in practice. For example, the MLF type system is simply not feasible for languages such as Haskell if it does not support qualified types properly, because of the prominent position of type classes in the Haskell language.

However, qualified types have several applications beyond type classes, and the fact that MLF proves to be compatible with qualified types says that MLF is a good underlying system for all languages that support any of these features. For instance, MLF the basis of the type system for the experimental Morrow language [7], which deals with records and employs MLF in the expectation that it can use an efficient type-directed translation where predicates correspond to runtime offsets [1].

Furthermore, we show that adding qualified types to MLF is not just useful in its own right but can also profit from impredicative types itself. This can lead to new applications of qualified types that are not possible with current predicative type systems.

### 3.1 Predicates

Qualified types extend the type language with *predicates*. A qualified type $\forall\pi.\,\sigma$ denotes those instances of $\sigma$ that satisfy the predicate $\pi$. For consistency with MLF we write predicates in the same way as bounds (instead of the more usual $\pi \Rightarrow \sigma$). The theory of qualified types makes just few assumptions of the language of predicates $\pi$, and there are many interesting instances. We discuss three of those extensions in this paper. One of the most widely known is the type class system in Haskell.

### 3.2 Type classes

A type class in Haskell denotes a family of types (*instances*) on which a number of values (the *member functions*) are defined. Each predicate $C\ \tau$ is an assertion that $\tau$ is an instance of class $C$. For example, the class $Eq$ denotes those types for which equality $(==)$ is defined:

$$(==) :: \forall\alpha\,(Eq\ \alpha).\,\alpha \to \alpha \to Bool$$

The predicate $Eq\ \alpha$ indicates that equality is not parametrically polymorphic, but only works for those types that are an instance of the $Eq$ class. In other words, type classes implement ad-hoc overloading where functions can behave differently for different types. This normally requires some sort of global analysis. By adding predicates to the types, this analysis is provided automatically by the theory of qualified types: constraints are specified locally in the predicates of a type, and these predicates are propagated through expressions by the type system. Take for instance the following expression:

$$(\lambda\alpha\ \beta.\ \mathbf{if}\ (\alpha == \beta)\ \mathbf{then}\ \texttt{"yes"}\ \mathbf{else}\ \texttt{"no"})$$
$$:: \forall\alpha\,(Eq\ \alpha).\,\alpha \to \alpha \to String$$

The equality predicate of $(==)$ is automatically propagated to the type of the entire expression. Operationally, we can interpret predicates as extra runtime parameters that give the runtime evidence that the predicate holds. For an $Eq\ \alpha$ predicate, this is would for example be a dictionary that contains the implementation of the equality function for type $\alpha$.

It is not advisable to allow type schemes in type class predicates themselves as instance resolution would become much more complicated, or even undecidable. However, the combination of MLF with type classes can still lead to new uses of overloading. This is mostly because polymorphic functions become more useful. Take for example a plain MLF list that contains the identity function:

$$xs :: [\forall\alpha.\, \alpha \to \alpha]$$
$$xs = [id]$$

This list is not very exciting as there simply exist few functions that have such a polymorphic type (modulo undefined values, just the identity function). With a type class, we can assert that we can apply certain operations to a polymorphic type. Here is a more interesting example with a list of functions that work on any numeric type:

$$\begin{array}{ll}\textbf{let} & precise :: (\forall\alpha\,(Num\ \alpha).\, \alpha \to \alpha) \to Bool \\ & \quad precise\ f = (f\ 1 \neq round\ (f\ 0.9)) \\ \textbf{in} & map\ precise\ [id, (+1), negate, (*2)]\end{array}$$

This is also an example of how MLF scales with respect to predicative systems: we can reuse the $map$ abstraction on lists with polymorphic components without having to resort to packing and unpacking (cf. Section 2.4).

### 3.3 Implicit parameters

Another instance of qualified types are implicit parameters. A predicate $?x :: \tau$ asserts that the term can access an implicit argument $?x$ with type $\tau$. This reduces the burden on a programmer to pass this argument explicitly. A typical application for implicit parameters are options that can be used inside deeply nested functions without passing them explicitly through each call site. For example, a pretty printer may use the width of the screen deep inside the render function:

$$pretty :: \forall(?width :: Int).\, Doc \to String$$
$$pretty\ doc = \ldots\ \textbf{if}\ (i\ <\ ?width)\ \ldots$$

The type signature of $pretty$ asserts that it expects an implicit argument $width$ of type $Int$, and a normal argument of type $Doc$. Implicit arguments are bound using the **with** keyword:

$$(pretty\ (text\ \texttt{"hi"})\ \textbf{with}\ ?width = 78) :: String$$

An interesting advantage with respect to type classes is that we can bind an implicit argument of a certain type to different values, while type classes only allow a single instance per type.

Of course, we may want to bind implicit arguments to polymorphic values. For example, we could pass monad operations as an implicit argument instead of using overloading, as is standard in Haskell. The function $twice$ applies an implicit $unit$ parameter to its argument:

$$twice :: \forall m\,(?unit :: \forall\beta.\, \beta \to m\ \beta)\,\alpha.\, \alpha \to m\ (m\ \alpha)$$
$$twice\ x = ?unit\ (\,?unit\ x)$$

Note that we use $?unit$ at two different types and that the type of $?unit$ must be polymorphic. Alas, the function $twice$ will not type check in a predicative type system as the type of the implicit parameter predicate is instantiated to a type scheme. Here we see that the combination of an impredicative type system like MLF and qualified types can lead to new applications of qualified types themselves. In the next section, we give another example of this in the context of records.

### 3.4 Records

Record operations can also be elegantly typed with qualified types. For example, the *has* predicate $(l :: \tau \in r)$ asserts that a record $r$ contains a particular field $l$ of type $\tau$. Using this predicate, we can give a type signature for record selection:

$$(\_.l) :: \forall \alpha\, r\, (l :: \alpha \in r).\, r \to \alpha$$

The qualified type allows us to write functions that work for any record containing a particular field. For example, the function $len$ works for any record containing an $x$ and $y$ field of type $Float$.

$$len :: \forall r\, (x :: Float \in r)\, (y :: Float \in r).\, r \to Float$$
$$len\ r = sqrt\ (r.x * r.x + r.y * r.y)$$

Polymorphic predicates arise naturally with records. For example, the type signature for monads $m$ contains two polymorphic fields:

$$\begin{array}{l}\textbf{type}\ Monad\ m = \\ \quad \{\, unit :: \forall \alpha.\quad \alpha \to m\ \alpha \\ \quad ,\, bind :: \forall \alpha\, \beta.\, m\ \alpha \to (\alpha \to m\ \beta) \to m\ \beta\}\end{array}$$

We can now define the function $twice$ with an explicit monad record as its argument, which applies the $unit$ field twice on its second argument:

$$twice :: \forall m\, (Monad\ m).\, Monad\ m \to (\forall \alpha.\, \alpha \to m\ (m\ \alpha))$$
$$twice\ r\ x = r.unit\ (r.unit\ x)$$

As before, $twice$ uses the $unit$ field at two different types. The selection of $unit$ on $r$ gives rise to the predicate $unit :: \forall \alpha.\, \alpha \to m\ \alpha \in Monad\ m$. This predicate obviously holds, but it is just not allowed in a predicative type system: the field type of the *has* predicate is instantiated here to a type scheme, and thus requires impredicative predicates. Record systems based on *lacks* predicates [1] suffer from the same problem as the record type in the predicate is instantiated with a record containing type schemes.

### 3.5 Qualified types and MLF

Since MLF is already based on bounded quantification, it is relatively easy to extend the theory of MLF with qualified types. First of all, we allow predicates to occur together with the bounds in a prefix. Besides extending the equivalence relation we just need to add two new instance rules for predicates:

$$(Q)\ \sigma \sqsubseteq \forall \pi.\, \sigma \qquad \frac{Q \Vdash \pi}{(Q)\ \forall \pi.\, \sigma \sqsubseteq \sigma}$$

The first rule states that we make a type less polymorphic by adding a qualifier. The second rule goes in the other direction: if a predicate is entailed by the context, we can leave it out. Note that we generalize the entailment relation of the theory of qualified types to work under a prefix $Q$ instead of a set of predicates.

## 4. Evidence translation

Qualified types are usually implemented using a technique called *evidence translation*. During evidence translation, implicit information represented by the predicates is turned into explicit function arguments. The compiler automatically abstracts from and provides evidence where necessary. In Section 4.1, we show how to use evidence translation for different sorts of qualified types. In Section 4.2, we discuss why a naïve evidence translation for MLF fails. We then describe how to overcome this problem in Section 4.3. This section contains the core idea of our translation, which is made precise in Section 5.

$$
\begin{aligned}
xs_1^* &= [\,]^* \\
xs_2^* &= \Lambda\gamma.\, \lambda(v_2 :: (\forall\alpha\ \beta \cdot \alpha \to \beta \to \alpha) \to \gamma).\, v_2\ const^* : xs_1^*\ \gamma \\
xs_3^* &= \Lambda\gamma.\, \lambda(v_3 :: (\forall\alpha\ (Ord\ \alpha) \cdot \alpha \to \alpha \to \alpha) \to \gamma).\, v_3\ min^* : xs_2^*\ \gamma\ (\lambda x.\, v_3\ (\Lambda\alpha.\, \lambda ord_\alpha.\, x\ \alpha\ \alpha)) \\
xs_4^* &= (<)^*\ Bool\ ord_{Bool} : xs_3^*\ \gamma\ (\lambda x.\, x\ Bool\ ord_{Bool})
\end{aligned}
$$

**Figure 2.** Idea of the evidence translation for MLF with qualified types

### 4.1 Examples of evidence translation

All three of the applications of qualified types discussed in the previous section can be translated in this fashion.

With type classes, the class constraints are turned into *dictionary arguments*. Dictionaries are records containing all the methods of a class. For instance, a dictionary for $Eq\ Int$ contains the equality and unequality functions for $Int$, because the $Eq$ class in Haskell has precisely these two methods.

Whenever a let-defined value is overloaded, i.e., has a class constraint, its translation expects an additional argument, namely the dictionary. And whenever an overloaded function is called, the compiler supplies the appropriate dictionary argument.

For implicit parameters, the situation is simpler than for type classes, as the evidence consists of only a single value, the implicit parameter itself.

When dealing with records, the evidence of a *has* constraint of the form $(l :: \tau \in r)$ can be the offset of label $l$ in record $r$.

In the following, we will investigate how we can perform evidence translation in a system based on MLF.

### 4.2 Evidence in MLF

When adding qualified types to MLF, it is relatively straightforward to extend the type rules in order to deal with predicates. However, higher rank and impredicativity makes it non-trivial to perform evidence translation.

Look at the following four lists:

$$
\begin{aligned}
xs_1 &= [\,] && :: \forall\alpha.\, [\alpha] \\
xs_2 &= const : xs_1 && :: [\forall\alpha\,\beta.\, \alpha \to \beta \to \alpha] && \text{-- } const :: \forall\alpha\,\beta.\, \alpha \to \beta \to \alpha \\
xs_3 &= min : xs_2 && :: [\forall\alpha\,(Ord\ \alpha).\, \alpha \to \alpha \to \alpha] && \text{-- } min :: \forall\alpha\,(Ord\ \alpha).\, \alpha \to \alpha \to \alpha \\
xs_4 &= (<) : xs_3 && :: [Bool \to Bool \to Bool] && \text{-- } (<) :: \forall\alpha\,(Ord\ \alpha).\, \alpha \to \alpha \to Bool
\end{aligned}
$$

Each of the lists is obtained from the previous one by adding one additional element in the front, using the cons operation $(:)$. With each of the additions, the type of the list changes.

All the lists contain elements that are polymorphic. The types are thus different from the types that Haskell would assign. In fact, the Haskell types of the four lists would be

$$
\begin{aligned}
&xs_1 :: \forall\alpha.\, [\alpha] \\
&xs_2 :: \forall\alpha\,\beta.\, [\alpha \to \beta \to \alpha] \\
&xs_3 :: \forall\alpha\,(Ord\ \alpha).\, [\alpha \to \alpha \to \alpha] \\
&xs_4 :: [Bool \to Bool \to Bool]
\end{aligned}
$$

all instances of the corresponding MLF types. However, the Haskell types of $x_2$ and $x_3$ are strictly less general than their MLF counterparts. With the MLF types, we can extract elements from the lists and use them polymorphically; with the Haskell types, we cannot. Consider a function

$$\mathbf{let}\ f\ ys = (head\ ys\ 2\ 3, head\ ys\ \texttt{'a'}\ \texttt{'b'})\ \mathbf{in}\ f\ xs_2$$

as an example of a function which could not be typed (be it annotated or not) in Haskell, because it really requires the more general impredicative MLF type of $xs_2$.

The evidence translation of each of the lists in Haskell is easy, because evidence is only passed on the outside of a let-bound term. We can represent the lists at runtime as follows:

$$xs_1^* = [\,]^*$$
$$xs_2^* = \Lambda\alpha\ \beta.\, const^*\ \alpha\ \beta : xs_1^*\ (\alpha \to \beta \to \alpha)$$
$$xs_3^* = \Lambda\alpha.\, \lambda ord_\alpha.\, min^*\ \alpha\ ord_\alpha : xs_2^*\ \alpha\ \alpha$$
$$xs_4^* = (<)^*\ Bool\ ord_{Bool} : xs_3^*\ Bool\ ord_{Bool}$$

Here, $x^*$ denotes the runtime term corresponding to the source term $x$. We are using System F as the runtime language in this paper, because it is sufficiently powerful to express the programs we are interested in, and it is fully typed; it is thus easy to see that the evidence translation produces well-typed terms. To retain clarity, however, we have omitted the type arguments of the $(:)$ calls as well as the annotations for the dictionary arguments, because they are uninteresting to the example at hand.

This runtime representation is, however, not adequate if we want to use the MLF types above, because in order to extract a value from such a list, we must *first* provide evidence fixing the type and the dictionary. This way, $xs_2$ cannot be used in function $f$ above.

What we want is a representation of the lists where each element accepts the evidence, "within" the list. For instance,

$$xs_2^* = [const^*]$$
$$xs_3^* = [\Lambda\alpha.\, \lambda ord_\alpha.\, min^*\ \alpha\ ord_\alpha, \Lambda\alpha.\, \lambda ord_\alpha.\, const^*\ \alpha\ \alpha]$$

In $xs_3$, both elements now accept evidence for $Ord\ \alpha$, but only $min$ makes use of it. All list elements must have the same runtime representation, because they are of the same type.

But $xs_3 = min : xs_2$, so there must be a way to construct $xs_3^*$ from $xs_2^*$! Similarly, we have to construct $xs_4^*$, which as a monomorphic list should still have the same representation as given above, from $xs_3^*$. In this simple situation, we could map over $xs_2$, adding unused abstractions over $ord_\alpha$, and over $xs_3$, providing evidence for $Ord\ \alpha$ by supplying the $ord_{Bool}$ dictionary. But in the general case, very complicated traversals of runtime values might be required, that are both difficult to get correct and inefficient to perform. Interestingly, Peyton Jones and Shields have identified this problem in their discussion of design decisions, and argue that the necessity of such traversals makes impredicative datatypes infeasible [10, Section 7.3].

### 4.3 Evidence translation using transformation functions

We are now going to present a variation of the standard evidence translation that overcomes the problem of traversing data structures in complicated ways at runtime. The idea is to build the necessary flexibility into the runtime representation from the beginning. The core idea of our evidence translation for MLF is that polymorphic values are always applied to a transformation function over which we can abstract.

If we later learn more about the polymorphic type, we can supply a suitable transformation function. If we learn enough about a value that it becomes monomorphic, we can supply a transformation function which removes all polymorphism. Often, however, we gain only partial information about a polymorphic type, such that two quantified variables must be the same. In such a case we supply a transformation function which applies the knowledge gained and subsequently calls a new transformation function. In effect, we thus substitute one transformation with a new one, more specific than the first.

Applying this idea, the runtime representations of the four lists become as shown in Figure 2. Again, we have omitted irrelevant annotations for $(:)$ and dictionary arguments. To understand these representations, it is useful to look at the real MLF types of the lists, with the presentation convention that we defined in Section 2.5 stripped:

$$xs_1 :: \forall(\alpha \geq \bot).\, [\alpha]$$
$$xs_2 :: \forall(\gamma \geq \forall\alpha\,\beta.\, \alpha \to \beta \to \alpha).\, [\gamma]$$

$$
\begin{array}{lll}
\tau^* & \doteq \tau & \\
\bot^* & \doteq \forall\alpha\cdot\alpha & \\
(\forall(\alpha \diamond_v \sigma_\alpha).\,\sigma)^* & \doteq \sigma^* & \text{if } \alpha \notin \mathsf{ftv}(\sigma) \\
(\forall(\alpha \diamond_v \sigma_\alpha).\,\sigma)^* & \doteq \sigma^*_\alpha & \text{if } \mathsf{nf}(\sigma) = \alpha \\
(\forall(\alpha \diamond_v \sigma_\alpha).\,\sigma)^* & \doteq \sigma^*\,[\alpha \mapsto \tau] & \text{if } \mathsf{nf}(\sigma_\alpha) = \tau \\
(\forall(\alpha \diamond_v \bot).\,\sigma)^* & \doteq \forall\alpha\cdot\sigma^* & \\
(\forall(\alpha \diamond_v \sigma_\alpha).\,\sigma)^* & \doteq \forall\alpha\cdot(\sigma^*_\alpha \to \alpha) \to \sigma^* &
\end{array}
$$

**Figure 3.** Translation of types

$$
\begin{array}{l}
xs_3 :: \forall(\gamma \geq \forall\alpha\,(Num\ \alpha).\,\alpha \to \alpha \to \alpha).\,[\gamma] \\
xs_4 :: [Bool \to Bool \to Bool]
\end{array}
$$

A non-trivial flexible bound $(\gamma \geq \sigma)$ is at runtime represented by an additional parameter of type $\mathsf{tr}(\sigma) \to \gamma$, where $\mathsf{tr}(\sigma)$ is the runtime representation of $\sigma$. This parameter can be used to add, remove, or modify evidence deep down in the value, as is required by the MLF type system. In particular, from $xs_2$ to $xs_3$, we transform $xs_2$ to accept evidence for $Ord\ \alpha$ which is ignored. And from $xs_3$ to $xs_4$ we pass evidence for $Ord\ Bool$ to $xs_3$. These two steps correspond to the instance rules that we have given for qualified types in Section 3.5. Note that the list $xs_4$ is no longer polymorphic, and does thus not expect any further transformation function.

In this simple example, the passing of transformation functions essentially amounts to mapping over the list multiple times, but in the general situation, we can apply exactly the same technique to perform complex transformations at low cost.

Note that we are actually only interpreting the concept of evidence translation in a more rigorous way. The additional transformation functions that are passed around are evidence on their own: evidence for the fact that quantified types indeed respect their bounds!

The actual translation that we introduce in the following section passes a few more arguments for technical reasons, which will, however, all be instantiated to identity functions. A clever compiler can easily optimize the generated expressions statically to pass evidence and evidence transformers only where they are actually needed, i.e., where bounded quantifications occur in the types.

## 5. Type inference for MLF

In this section, we present the type-inference algorithm taken from the MLF paper [6], augmented with a type-directed translation that produces a System-F term (also called *runtime term*) from an MLF term.

### 5.1 Runtime types

The difference between a term and its translation is that the translation is fully type-annotated, and that evidence is passed explicitly. Since the evidence that is required is dictated by the MLF types, it is not surprising that the types of the translated System-F terms are directly related to the types of the original MLF terms.

Figure 3 shows how a System-F type $\sigma^*$ can be computed from an MLF type $\sigma$. We call $\sigma^*$ the *runtime type* of $\sigma$. The definition is organized in such a way that the following property holds:

THEOREM 2. *The runtime type of an MLF type $\sigma$ is the runtime type of its normal form*[2]*, i.e.,*

$$\sigma^* = (\mathsf{nf}(\sigma))^*$$

The translation of a monotype $\tau$ is $\tau$ itself. No qualified types can occur in monotypes, neither is there polymorphism, so we can safely reuse $\tau$, which is a valid System-F type. The type $\bot$ is mapped to $\forall\alpha\cdot\alpha$.

[2] Recall that Figure 1 describes how to compute the normal form of an MLF type.

When dealing with bounds, most cases are dictated by the desired property given by Theorem 2. The next three cases are therefore directly based on the corresponding rules for normalization.

A constraint for a variable that does not occur in the rest of the type is irrelevant and can be dropped. No evidence is required, because it would not be used anyway. If we have a type $\forall(\alpha \diamond_v \sigma_\alpha).\, \sigma$ where $\mathsf{nf}(\sigma) = \alpha$, then the type is equivalent to $\sigma_\alpha$, and we can use the runtime representation of $\sigma_\alpha$. Each monotype $\tau$ in MLF has the property that only $\tau$ itself is an instance of $\tau$. Since we already argued that there is no need for evidence in relation to monotypes, we can inline a constraint where the bound is equivalent to a monotype.

The final two cases deal with non-trivial bounds. In the case of an unbounded quantification, we introduce a quantification in System F as well. Because any type argument will do, no further information is required. This is different in the final case:

$$(\forall(\alpha \diamond_v \sigma_\alpha).\, \sigma)^* \doteq \forall\alpha \cdot (\sigma_\alpha^* \to \alpha) \to \sigma^*$$

It shows that any bound that does not match any of the other cases is represented using a function argument. The argument provides evidence for the bound: for a flexible bound, it demonstrates that $\alpha$ is indeed an instance of $\sigma_\alpha$ by giving a transformation from any $\sigma_\alpha^*$ value into an $\alpha$ value; for a rigid bound, the argument demonstrates that $\alpha$ is equivalent to $\sigma_\alpha$.

For the type inference algorithm to be correct, it must supply well-behaved functions for evidence parameters. The transformation functions may provide, remove, or reorder evidence, but not add any computation beyond that. As an example of the type translation, consider the type of $xs_2$ from Section 4:

$$\forall(\gamma \geq \forall\alpha\,\beta.\, \alpha \to \beta \to \alpha).\, [\gamma]$$

Let $\sigma_\gamma = \forall\alpha\,\beta.\, \alpha \to \beta \to \alpha$. Then $\forall(\gamma \geq \sigma_\gamma).\, [\gamma]$ matches the last case in the definition of $(\_)^*$, hence its translation is

$$\forall\gamma \cdot (\sigma_\gamma^* \to \gamma) \to [\gamma]^*$$

The type $[\gamma]$ is a monotype, its translation is thus also $[\gamma]$. It remains to translate $\sigma_\gamma$. The quantifications on $\alpha$ and $\beta$ are unbounded, and abbreviations for $(\alpha \geq \bot)$ and $(\beta \geq \bot)$. The next to last case matches twice, hence $\sigma_\gamma^* = \forall\alpha\ \beta \cdot \alpha \to \beta \to \alpha$, and consequently

$$\forall\gamma \cdot ((\forall\alpha\ \beta \cdot \alpha \to \beta \to \alpha) \to \gamma) \to [\gamma]$$

If we return to the runtime representation given for $x_2$ in Figure 2, namely

$$\Lambda\gamma.\, \lambda(v_2 :: (\forall\alpha\ \beta \cdot \alpha \to \beta \to \alpha) \to \gamma).\, (v_2\ const^* : xs_1^*\ \gamma)$$

we see that it indeed has precisely this System-F type.

In the term translation we refer to runtime evidence by name, like the $v_2$ in the above example. In the translation algorithm we therefore make use of labeled prefixes where each bound is labeled. The runtime evidence for a bound $(\alpha \diamond_v \sigma)$ is now defined by the name $v$ with type $(\sigma^* \to \alpha)$. Of course, the labels are assigned internally by the compiler and are never exposed to the user. Furthermore, just like the names of quantifiers can be alpha-converted within a polytype $\sigma$, the labels associated with the bounds of the quantifiers can be converted as well, and we assume that this is implicitly done in such a way that all bound variables and all labels in a prefix are always distinct.

A slight complication to the translation algorithm is that we do not pass evidence for trivial bounds that are removed by normalizing the type. The type inference algorithm therefore makes use of two helper functions app and abs, given in Figure 4, that supply evidence parameters or abstract from evidence. Both functions turn an MLF type into a context that can be filled with a proper System-F term.

The function app supplies evidence arguments for a term of type $\sigma$. It is structured exactly like the cases of type translation and it behaves uniformly over normalized types:

$$
\begin{array}{lll}
\mathsf{app}(\tau) & \doteq \bullet & \\
\mathsf{app}(\bot) & \doteq \bullet\ (\forall\alpha\cdot\alpha) & \\
\mathsf{app}(\forall(\alpha \diamond_v \sigma_\alpha).\,\sigma) & \doteq \mathsf{app}(\sigma) & \text{if } \alpha \notin \mathsf{ftv}(\sigma) \\
\mathsf{app}(\forall(\alpha \diamond_v \sigma_\alpha).\,\sigma) & \doteq \mathsf{app}(\sigma_\alpha) & \text{if } \mathsf{nf}(\sigma) = \alpha \\
\mathsf{app}(\forall(\alpha \diamond_v \sigma_\alpha).\,\sigma) & \doteq \mathsf{app}(\sigma) & \text{if } \mathsf{nf}(\sigma_\alpha) = \tau \\
\mathsf{app}(\forall(\alpha \diamond_v \bot).\,\sigma) & \doteq \mathsf{app}(\sigma)\ [\bullet\ \alpha] & \\
\mathsf{app}(\forall(\alpha \diamond_v \sigma_\alpha).\,\sigma) & \doteq \mathsf{app}(\sigma)\ [\bullet\ \alpha\ v] &
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{abs}(\tau) & \doteq \bullet & \\
\mathsf{abs}(\bot) & \doteq \Lambda\alpha.\ \bullet & \\
\mathsf{abs}(\forall(\alpha \diamond_v \sigma_\alpha).\,\sigma) & \doteq \mathbf{let}\ \alpha \mapsto \sigma^*_\alpha; v \mapsto I\ \alpha\ \mathbf{in}\ \mathsf{abs}(\sigma) & \text{if } \alpha \notin \mathsf{ftv}(\sigma) \\
 & \doteq \mathbf{let}\ \alpha \mapsto \sigma^*_\alpha; v \mapsto I\ \alpha\ \mathbf{in}\ (\mathsf{abs}(\sigma_\alpha)[\mathsf{abs}(\sigma)]) & \text{if } \mathsf{nf}(\sigma) = \alpha \\
 & \doteq \mathbf{let}\ \alpha \mapsto \sigma^*_\alpha; v \mapsto I\ \alpha\ \mathbf{in}\ \mathsf{abs}(\sigma) & \text{if } \mathsf{nf}(\sigma_\alpha) = \tau \\
\mathsf{abs}(\forall(\alpha \diamond_v \bot).\,\sigma) & \doteq \Lambda\alpha.\ \mathbf{let}\ v \mapsto \lambda x.\,x\ \alpha\ \mathbf{in}\ \mathsf{abs}(\sigma) & \\
\mathsf{abs}(\forall(\alpha \diamond_v \sigma_\alpha).\,\sigma) & \doteq \Lambda\alpha.\,\lambda(v : \sigma^*_\alpha \to \alpha).\,\mathsf{abs}(\sigma) &
\end{array}
$$

**Figure 4.** Type-directed application and abstraction

$$\mathsf{app}(\sigma) = \mathsf{app}(\mathsf{nf}(\sigma))$$

Furthermore, we can easily check that it always supplies type-correct evidence for non-trivial bounds:

$$e^* :: (\forall Q.\,\tau)^* \Rightarrow \mathsf{app}(\forall Q.\,\tau)[e^*] :: \tau^*$$

The interesting cases for app are again the last two cases. For an unconstrained bound $(\alpha \diamond_v \bot)$ we pass the type parameter $\alpha$. This is the same as in a standard translation from Hindley-Milner types to System F. The last case supplies evidence for a non-trivial bound $(\alpha \diamond_v \sigma)$. As signified by the label, the evidence is bound in the environment by the name $v$, and we simply pass the type parameter $\alpha$ and the evidence $v$.

The abstraction function abs abstracts from evidence for non-trivial bounds, and binds the names and types of all trivial bounds. Again, this function is structured exactly like type translation and app. However, it does not behave uniformly over normalized types since it explicitly binds the names of non-trivial bounds that are eliminated during normalization. As we will see in the unification algorithm, the function abs is always used under a certain prefix. If a term $e$ has a translation term $e^*$ of type $\tau^*$ under a prefix $Q$, then abs can lift it to the type $(\forall Q.\,\tau)^*$. More formally:

$$\mathsf{abs}(\forall Q.\,\tau)[e^*] :: (\forall Q.\,\tau)^*$$

The abstraction algorithm needs to bind all evidence names and types in the prefix. Trivial bounds can be satisfied in only one way, we therefore substitute their labels and type arguments. It makes no sense to abstract over trivial evidence, we rather provide it immediately.

Non-trivial bounds, however, are lambda bound, and have to be supplied later.

The first two cases on monomorphic types and $\bot$ have nothing to abstract. The next three cases deal with trivial bounds. The first of these three deals with a dead binding $\alpha \notin \mathsf{ftv}(\sigma)$. It is safe to treat $\alpha$ as $\sigma_\alpha$, and to bind the evidence $v$ to an identity transformation. We use the notation $\mathbf{let}\ v \mapsto e\ \mathbf{in}\ x$ to denote the substitution of $v$ by $e$ in the System-F term $x$. The next case is a direct substitution where $\alpha$ can be bound to $\sigma_\alpha$ itself, and where $v$ is again an identity transformation. The last trivial bound concerns monomorphic types where the evidence can also be bound to an identity since $\tau^* = \tau$.

$$\text{(inf-var)}\quad \frac{x : \sigma \in \Gamma}{Q \,|\, \Gamma \vdash x \;:\; (Q, \sigma) \rightsquigarrow x}$$

$$\text{(inf-let)}\quad \frac{\begin{array}{c} Q \,|\, \Gamma \vdash e_1 \;:\; (Q_1, \sigma_1) \rightsquigarrow e_1^* \\ Q_1 \,|\, (\Gamma, x : \sigma_1) \vdash e_2 \;:\; (Q_2, \sigma_2) \rightsquigarrow e_2^* \end{array}}{\begin{array}{c} Q \,|\, \Gamma \vdash (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) \;:\; (Q_2, \sigma_2) \rightsquigarrow \\ (\lambda(x : \sigma_1^*).\ e_2^*)\ e_1^* \end{array}}$$

$$\text{(inf-app)}\frac{\begin{array}{c} \mathsf{fresh}(v, w, u, \alpha, \beta, \gamma) \\ Q \,|\, \Gamma \vdash e_1 \;:\; (Q_1, \sigma_1) \rightsquigarrow e_1^* \\ Q_1 \,|\, \Gamma \vdash e_2 \;:\; (Q_2, \sigma_2) \rightsquigarrow e_2^* \\ Q' = (Q_2, \alpha \geq_v \sigma_1, \beta \geq_w \sigma_2, \gamma \geq_u \bot) \\ (Q')\ \alpha \sim \beta \to \gamma \;:\; (Q_3, \mathcal{L}) \\ (Q_4, Q_5) = Q_3 \uparrow \mathsf{dom}(Q) \qquad \sigma = \forall Q_5.\, \gamma \\ e^* = \mathsf{abs}(\sigma)[\mathcal{L}\ \mathbf{in}\ (v\ e_1^*)\ (w\ e_2^*)] \end{array}}{Q \,|\, \Gamma \vdash (e_1\ e_2) \;:\; (Q_4, \sigma) \rightsquigarrow e^*}$$

$$\text{(inf-lam)}\frac{\begin{array}{c} \mathsf{fresh}(v, w, \alpha, \beta) \\ (Q, \alpha \geq_v \bot) \,|\, (\Gamma, x : \alpha) \vdash e \;:\; (Q_1, \sigma_1) \rightsquigarrow e^* \\ (Q_2, Q_3) = Q_1 \uparrow \mathsf{dom}(Q) \\ \sigma = \forall (Q_3, \beta \geq_w \sigma_1).\, \alpha \to \beta \\ f^* = \mathsf{abs}(\sigma)[\lambda(x : \alpha).\, w\ e^*] \end{array}}{Q \,|\, \Gamma \vdash (\lambda x.\, e) \;:\; (Q_2, \sigma) \rightsquigarrow f^*}$$

**Figure 5.** Type inference

The final two cases of abstraction handle non-trivial bounds where the evidence is passed at runtime. The unconstrained bound $(\alpha \diamond_v \bot)$ just abstracts over the type argument $\alpha$ – the runtime evidence $v$ is bound to a function that passes $\alpha$ as any type is trivially an instance of $\forall\alpha \cdot \alpha$. As apparent from the corresponding rule in app, the bound $(\alpha \diamond_v \sigma_\alpha)$ gets full evidence passed: both the type argument $\alpha$ and the runtime evidence $v$ are lambda bound, where $v$ is bound to a function that transforms a type $\sigma_\alpha^*$ to its instantiation type $\alpha$.

### 5.2 Type inference

We are now in the position to present the type inference algorithm for MLF, extended with type-directed translation, as shown in Figure 5. The expression $Q \,|\, \Gamma \vdash e \;:\; (Q', \sigma) \rightsquigarrow e^*$ infers for a given expression $e$, under a prefix $Q$ and type environment $\Gamma$ a type $\sigma$ that holds under prefix $Q'$. Furthermore, it derives a translated System-F term $e^*$. Note that the algorithm is exactly the same as the standard type inference algorithm of MLF modulo the translation terms.

THEOREM 3. *The type inference algorithm in Figure 5 derives a well-typed translation term:*

$$Q \,|\, \Gamma \vdash e \;:\; (Q', \sigma) \rightsquigarrow e^* \;\Rightarrow\; e^* :: \sigma^*$$

This can be proved by induction on the structure of types.

Usually, Hindley-Milner based type inference algorithms infer a type under a certain substitution. With MLF, the prefix subsumes the role of the substitution. In the prefix, type variables can be bound to monomorphic and polymorphic types. The type environment $\Gamma$ is standard and maps term variables $x$ to polymorphic types $\sigma$. The environment $\Gamma$ is extended with a variable $x$ as $\Gamma, x :: \sigma$, where the new binding of $x$ shadows a previous binding of $x$ in $\Gamma$. Initially, both the prefix $Q$ and the environment $\Gamma$ are empty.

The rule (inf-var) infers the type of a variable $x$ by simply looking up the type in the environment. The translated term is just $x$ and the prefix is unchanged. The rule (inf-let) is also straightforward: it infers the type for the **let** bound expression $e_1$ and then for the body $e_2$ with $x$ bound to the inferred type of $e_1$. The translated term is phrased as lambda expression since the System-F language has no **let** bindings. The rule for **let** bindings is much more involved in Hindley-Milner systems as it is the one place where generalization to polymorphic types takes place. In contrast, in MLF generalization is performed as part of the rules for application and lambda, hence no special actions are necessary for **let** bindings.

$$\text{(update)}\ \frac{(Q_1, (Q_3, \alpha \diamond_v \sigma_0, Q_4)) = Q \uparrow \mathsf{ftv}(\sigma) \quad (\diamond_v = (\geqslant)) \vee ((Q)\ \sigma_0 \sqsubseteq^? \sigma) \quad \mathsf{fresh}(w) \quad \mathcal{L}' = \mathbf{let}\ v \mapsto w \circ f\ \mathbf{in}\ \mathcal{L}}{(Q, \mathcal{L}) \lhd (\alpha \diamond \sigma)_f \doteq ((Q_1\, Q_3, \alpha \diamond_w \sigma, Q_4), \mathcal{L}')}$$

$$\text{(merge)}\ \frac{(\ (Q_1, \alpha_1 \diamond_{v_1} \sigma, Q_2, \alpha_2 \diamond_{v_2} \sigma, Q_3) = Q \vee (Q_1, \alpha_2 \diamond_{v_2} \sigma, Q_2, \alpha_1 \diamond_{v_1} \sigma, Q_3) = Q\ ) \quad Q' = (Q_1, \alpha_1 \diamond_{v_1} \sigma, \alpha_2 =_w \alpha_1, Q_2\, Q_3) \quad \mathsf{fresh}(w)}{(Q, \mathcal{L}) \lhd (\alpha_1 \wedge \alpha_2) \doteq (Q', \mathbf{let}\ v_2 \mapsto w \circ v_1\ \mathbf{in}\ \mathcal{L})}$$

**Figure 6.** Update and merge

The rule for application (inf-app) first infers the types for the function $e_1$ and the argument $e_2$. After that, it extends the prefix with fresh bindings for those types and unifies them using the unification algorithm shown in Figure 7. The unification algorithm is the core of MLF and we discuss this in depth in the next section. For now, it is sufficient to know that the expression $(Q)\ \tau_1 \sim \tau_2\ : (Q', \mathcal{L})$ unifies two monomorphic types $\tau_1$ and $\tau_2$ under prefix $Q$, returning a new prefix $Q'$ such that $(Q')\ \tau_1 \equiv \tau_2$. It also returns an *evidence substitution* $\mathcal{L}$ which is explained in the next paragraph. Since the MLF inference algorithm infers polymorphic types, it must generalize the result type $\gamma$. Generalization works by splitting the prefix resulting from unification ($Q_3$) under the domain of original prefix $Q$ in the prefixes $(Q_4, Q_5)$. The generalized type $\sigma$ of the application is simply $(\forall Q_5.\, \gamma)$ under the prefix $Q_4$. See Appendix A for a definition of the split function.

The translation of the application terms is more involved. First of all, we need to use the translated term $e_1^*$ and $e_2^*$ at their instantiated types, namely $\alpha$ and $\beta$. Fortunately, as apparent from the bounds in $Q'$, the runtime evidence for the conversion from $\sigma_1^*$ to $\alpha$ is given by $v$, and $w$ gives the evidence for $\sigma_2^* \to \beta$. A well-typed translated application is now given by $(v\ e_1^*)\ (w\ e_2^*)$. Of course, we must be careful to ensure that the names $v$ and $w$ are actually bound somewhere. As we saw in the previous section, the abstraction function binds all evidence of a certain $\sigma$. In our case, $\mathsf{abs}(\sigma)[(v\ e_1^*)\ (w\ e_2^*)]$ binds all evidence in $Q_5$. Since we work under prefix $Q_4$ all evidence in $Q_4$ and $Q_5$, i.e., $Q_3$ is bound. However, we took $v$ and $w$ from $Q'$ and we can not be sure that $Q_3$ is equal to $Q'$. That is where the evidence substitution $\mathcal{L}$ resulting from unification comes into play: it defines all evidence in $Q'$ in terms of evidence in $Q_3$. By applying this on the translated application, we get a well-formed System-F term: $\mathsf{abs}(\sigma)[\mathcal{L}\ \mathbf{in}\ (v\ e_1^*)\ (w\ e_2^*)]$.

The inference rule for lambda expression (inf-lam) infers a type for the body $e$ under the assumption that argument $x$ has type $\alpha$, where the bound on $\alpha$ is unconstrained $(\alpha \geq_v \bot)$. Again, it generalizes the result type by splitting the inferred prefix of the body $Q_1$ under the original prefix $Q$ in the prefixes $(Q_2, Q_3)$. The inferred type of the lambda expression $\sigma$ is now $\forall(Q_3, \beta \geq_w \sigma_1).\, \alpha \to \beta$. This nicely shows that the result of the lambda expression can be polymorphic itself. As a practical example, the inferred type of the *const* function in MLF is:

$$(\lambda x.\, \lambda y.\, x) :: \forall \alpha.\, \alpha \to (\forall \beta.\, \beta \to \alpha)$$

The translated term for a lambda expression uses the evidence $w$ to instantiate the term $e^*$ of type $\sigma_1^*$ to $\beta$, resulting in the term $(\lambda(x : \alpha).\, w\ \ e^*) :: (\alpha \to \beta)$. The abstraction function on $\sigma$ binds all evidence in $(Q_3, \beta \geq_w \sigma_1)$ and returns a well-formed System-F term of type $\sigma^*$, namely $\mathsf{abs}(\sigma)[\lambda(x : \alpha).\, w\ e^*]$.

### 5.3 Update and merge

Before we can describe the unification algorithm, we define two helper functions (update) and (merge) shown in Figure 6. These functions are used by unification and are the only functions that change the unification prefix. A consequence is that these are the only functions that change the evidence substitution. Note that up to the evidence substitution, the (update) and (merge) algorithms are exactly the same as the standard MLF algorithms.

$$\text{(uni-var)}\quad (Q)\ \alpha \sim \alpha\ : (Q, \epsilon)$$

$$\text{(uni-con)}\quad \frac{(Q_{i-1})\ \tau_i \sim \tau_i'\ : (Q_i, \mathcal{L}_i) \text{ for } i \in 1..n \quad \mathcal{L}' = \mathcal{L}_n \textbf{ in } \ldots \textbf{ in } \mathcal{L}_1}{(Q_0)\ g\ \tau_1 \ldots \tau_n \sim g\ \tau_1' \ldots \tau_n'\ : (Q_n, \mathcal{L}')}$$

$$\text{(uni-mvar-l)}\quad \frac{(\alpha \diamond_v \sigma) \in Q \quad \sigma \in \mathcal{V} \quad (Q)\ \tau \sim \mathsf{nf}(\sigma)\ : (Q', \mathcal{L})}{(Q)\ \tau \sim \alpha\ : (Q', \mathcal{L})}$$

$$\text{(uni-mvar-r)}\quad \frac{(\alpha \diamond_v \sigma) \in Q \quad \sigma \in \mathcal{V} \quad (Q)\ \mathsf{nf}(\sigma) \sim \tau\ : (Q', \mathcal{L})}{(Q)\ \alpha \sim \tau\ : (Q', \mathcal{L})}$$

$$\text{(uni-mono-r)}\quad \frac{(\alpha \diamond_v \sigma) \in Q \quad \tau \notin \mathcal{A} \quad \sigma \notin \mathcal{V} \quad \alpha \notin \mathsf{dom}(Q\ /\ \tau) \quad (Q)\ \sigma \sim_m \tau\ : (Q_0, \mathcal{L}_0, f) \quad (Q_1, \mathcal{L}_1) = (Q_0, \mathcal{L}_0) \lhd (\alpha \diamond \tau)_f}{(Q)\ \alpha \sim \tau\ : (Q_1, \mathcal{L}_1)}$$

$$\text{(uni-mono-l)}\quad \frac{(\alpha \diamond_v \sigma) \in Q \quad \tau \notin \mathcal{A} \quad \sigma \notin \mathcal{V} \quad \alpha \notin \mathsf{dom}(Q\ /\ \tau) \quad (Q)\ \sigma \sim_m \tau\ : (Q_0, \mathcal{L}_0, f) \quad (Q_1, \mathcal{L}_1) = (Q_0, \mathcal{L}_0) \lhd (\alpha \diamond \tau)_f}{(Q)\ \tau \sim \alpha\ : (Q_1, \mathcal{L}_1)}$$

$$\text{(uni-poly)}\quad \frac{(\alpha_1 \diamond_{v_1} \sigma_1) \in Q \quad (\alpha_2 \diamond_{v_2} \sigma_2) \in Q \quad \alpha_1 \neq \alpha_2 \quad \sigma_1 \notin \mathcal{V} \quad \sigma_2 \notin \mathcal{V} \quad \alpha_1 \notin \mathsf{dom}(Q\ /\ \sigma_2) \quad \alpha_2 \notin \mathsf{dom}(Q\ /\ \sigma_1) \quad (Q)\ \sigma_1 \sim_p \sigma_2\ : (Q_0, \sigma_0, \mathcal{L}_0, f_1, f_2) \quad (Q_1, \mathcal{L}_1) = (Q_0, \mathcal{L}_0) \lhd (\alpha_1 \diamond \sigma_0)_{f_1} \lhd (\alpha_2 \diamond \sigma_0)_{f_2} \lhd (\alpha_1 \wedge \alpha_2)}{(Q)\ \alpha_1 \sim \alpha_2\ : (Q_1, \mathcal{L}_1)}$$

**Figure 7.** Monomorphic unification

$$\text{(mono-bot)}\quad (Q)\ \bot \sim_m \tau\ : (Q, \epsilon, \lambda x.\ x\ \tau^*)$$

$$\text{(mono-poly)}\quad \frac{\forall Q_1.\ \tau_1 = \sigma_1 \quad (Q Q_1)\ \tau_1 \sim \tau\ : (Q_2, \mathcal{L}) \quad (Q_3, Q_4) = Q_2 \uparrow \mathsf{dom}(Q) \quad \sigma = \forall Q_4.\ \tau \quad f_{\sigma_1^* \to \tau^*} = \lambda e_{\sigma_1^*}.\ \mathsf{abs}(\sigma)[\mathcal{L} \textbf{ in } \mathsf{app}(\sigma_1)[e]]}{(Q)\ \sigma_1 \sim_m \tau\ : (Q_3, \mathcal{L}, f)}$$

**Figure 8.** Mono-poly unification

The expression $(Q, \mathcal{L}) \lhd (\alpha \diamond \sigma)_f$ updates the prefix $Q$ and evidence substitution $\mathcal{L}$ returning a pair $(Q', \mathcal{L}')$ where $Q'$ is the updated prefix, and where $\mathcal{L}'$ is the updated evidence substitution. As defined in Figure 6, the update operation updates the bound $(\alpha \diamond_v \sigma_0) \in Q$ with the bound $(\alpha \diamond_w \sigma)$. It also prevents the update of rigid bindings with a less polymorphic type through the *abstraction check* algorithm $(Q)\ \sigma_0 \sqsubseteq^? \sigma$, which is described in the thesis of Le Botlan [5].

Since the bound of $v$ disappears from the prefix, we need to bind it in the evidence substitution $\mathcal{L}$. The evidence term for $v$ must have type $\sigma_0^* \to \alpha$. Since the newly bound evidence $w$ has type $\sigma^* \to \alpha$, we just need a runtime function of type $\sigma_0^* \to \sigma^*$ to be able to bind the evidence $v$ in terms of the new prefix. This function $f$ is passed with the update expression as a subscript of the new bound and will be constructed during unification. Therefore, we can substitute all occurrences of evidence $v$ by the term $w \circ f$.

The merge expression $(Q, \mathcal{L}) \lhd (\alpha_1 \wedge \alpha_2)$ merges the equal bounds of two type variables $\alpha_1$ and $\alpha_2$ in the prefix $Q$ and evidence substitution $\mathcal{L}$, returning an updated prefix and evidence substitution as a pair $(Q', \mathcal{L}')$. Since the evidence $v_2$ disappears, we redefine it in terms of the new evidence $w$ and $v_1$. Since the evidence $w$ transforms $\alpha_1 \to \alpha_2$ and $v_1$ transforms $\sigma^* \to \alpha_1$, we can substitute all occurrences of the evidence $v_2$ by the term $w \circ v_1$.

### 5.4 Unification

The algorithm for unification ($\sim$) is defined in Figure 7 and makes use of the helper functions for mono-poly unification ($\sim_m$) defined in Figure 8 and poly unification ($\sim_p$) defined in Figure 9. Again, the unification

$$\text{(poly-bot-l)}\quad (Q)\ \bot \sim_p \sigma\ : (Q, \sigma, \epsilon, \lambda x.\, x\ \sigma^*, I\ \sigma^*)$$

$$\text{(poly-bot-r)}\quad (Q)\ \sigma \sim_p \bot\ : (Q, \sigma, \epsilon, I\ \sigma^*, \lambda x.\, x\ \sigma^*)$$

$$\text{(poly-poly)}\quad \frac{\begin{array}{c}\forall Q_1.\, \tau_1 = \sigma_1 \quad \forall Q_2.\, \tau_2 = \sigma_2 \\ \mathsf{disjoint}(\mathsf{dom}(Q), \mathsf{dom}(Q_1), \mathsf{dom}(Q_2)) \\ (Q Q_1 Q_2)\ \tau_1 \sim \tau_2\ : (Q_0, \mathcal{L}) \\ (Q_3, Q_4) = Q_0 \uparrow \mathsf{dom}(Q) \quad \sigma = \forall Q_4.\, \tau_1 \\ f_{\sigma_1^* \to \sigma^*} = \lambda e_{\sigma_1^*}.\, \mathsf{abs}(\sigma)[\mathcal{L}\ \mathbf{in}\ \mathsf{app}(\sigma_1)[e]] \\ g_{\sigma_2^* \to \sigma^*} = \lambda e_{\sigma_2^*}.\, \mathsf{abs}(\sigma)[\mathcal{L}\ \mathbf{in}\ \mathsf{app}(\sigma_2)[e]]\end{array}}{(Q)\ \sigma_1 \sim_p \sigma_2\ : (Q_3, \sigma, \mathcal{L}, f, g)}$$

**Figure 9.** Poly unification

algorithm is exactly like that of MLF modulo the evidence translation. Also, we have specialized the poly unification where one argument is a mono type to the mono-poly unification for ease of presentation.

As stated before, the expression $(Q)\ \tau_1 \sim \tau_2\ : (Q', \mathcal{L})$ unifies two monomorphic types $\tau_1$ and $\tau_2$ under prefix $Q$, returning a new prefix $Q'$ such that $\tau_1$ and $\tau_2$ are equivalent under $Q'$, i.e., $(Q')\ \tau_1 \equiv \tau_2$. The evidence substitution $\mathcal{L}$ binds all evidence of $Q$ in terms of evidence in $Q'$. The unification essentially follows the structure of standard first-order unification, except that computation of the unifier is replaced by the computation of a unifying prefix. Furthermore, we need to do extra work for variables bound to polymorphic types.

During unification, we frequently have to perform a kind of "occurs check", using the notation $\alpha \notin \mathsf{dom}(Q\ /\ \sigma)$, where $\mathsf{dom}(Q\ /\ \sigma)$ is defined in Appendix A.

The first rule states that equal variables unify with an unchanged prefix and an empty evidence substitution. Constructors unify if all their arguments unify. The rules (uni-mvar-l) and (uni-mvar-r) unify bounds that are variables themselves. The next two rules, (uni-mono-l) and (uni-mono-r) unify monomorphic types with a possibly polymorphic bound using mono-poly unification and update the bound of $\alpha$ to the monomorphic type $\tau$. The last rule unifies two polymorphic bounds using poly unification, updating and merging the bounds of $\alpha_1$ and $\alpha_2$ with the possibly polymorphic type $\sigma$ that is a common instance of their bounds.

The poly unification ($\sim_p$) algorithm is defined in Figure 9. The expression $(Q)\ \sigma_1 \sim_p \sigma_2\ : (Q', \sigma, \mathcal{L}, f, g)$ unifies two polymorphic types $\sigma_1$ and $\sigma_2$ under prefix $Q$. The algorithm assumes that $\sigma_1$ and $\sigma_2$ are in constructed form. The constructed form is a weak form of a normalization that just reveals the structure of polymorphic type [6], given in Appendix A. Poly unification returns a new prefix $Q'$ under which the type $\sigma$ is a common instance of $\sigma_1$ and $\sigma_2$. Furthermore, poly unification returns an evidence substitution and two translations functions: $f$ of type $\sigma_1^* \to \sigma^*$, and $g$ of type $\sigma_2^* \to \sigma^*$. The latter functions are used by the (uni-poly) rule to update the bounds. The first two rules (poly-bot-l) and (poly-bot-r) unify with unconstrained bounds and return trivial transformations. Things become more interesting in rule (poly-poly) where two non-trivial polymorphic types are unified.

The algorithm is exactly like that of MLF: first it instantiates both types and than generalizes over the result. The interesting part is formed by the construction of the evidence transformers $f$ and $g$. The transformer $f$ must have the runtime type $\sigma_1^* \to \sigma^*$, and thus takes a runtime term $e$ of type $\sigma_1^*$. The term is instantiated to type $\tau_1$ by using the $\mathsf{app}(\sigma_1)$ function. After binding the evidence under $Q_0$ using the evidence substitution $\mathcal{L}$ resulting from the mono unification, we can use abstraction over $\sigma$ to transform to a runtime term of type $\sigma^*$, namely $f_{\sigma_1^* \to \sigma^*} = \lambda e_{\sigma_1^*}.\, \mathsf{abs}(\sigma)[\mathcal{L}\ \mathbf{in}\ \mathsf{app}(\sigma_1)[e]]$. The construction of $g$ is equivalent. Note that we rely essentially here on the law that $\mathsf{app}(\sigma) = \mathsf{app}(\mathsf{nf}(\sigma))$, otherwise we could not work on the constructed forms required by the poly unification.

The mono-poly unification is basically just a specialization of poly unification where one argument is a monomorphic type. Just like poly unification, it expects its arguments in constructed form. Instead of two functions that transform to a common instantiation, mono-poly unification just returns a single evidence transformer $f$ of type $\sigma^* \to \tau^*$, which is the main reason for creating a specialized instance of poly unification.

$$(\forall(\pi_v).\,\sigma)^* \quad \doteq \pi^* \to \sigma^*$$
$$\mathsf{app}(\forall(\pi_v).\,\sigma) \doteq \mathsf{app}(\sigma)\ [\bullet\ v]$$
$$\mathsf{abs}(\forall(\pi_v).\,\sigma) \doteq \lambda(v : \pi^*).\,\mathsf{abs}(\sigma)$$

**Figure 10.** Qualified type translation, application, and abstraction

$$\mathsf{simplify}(Q, \tau) \doteq (\tau, \epsilon)$$
$$\mathsf{simplify}(Q, \bot) \doteq (\bot, \epsilon)$$
$$\frac{Q \Vdash \pi \rightsquigarrow e^* \quad (\sigma', \mathcal{L}) = \mathsf{simplify}(Q, \sigma)}{\mathsf{simplify}(Q, \forall(\pi_v).\,\sigma) \doteq (\sigma', \mathbf{let}\ v \mapsto e^*\ \mathbf{in}\ \mathcal{L})}$$
$$\frac{(\sigma', \mathcal{L}) = \mathsf{simplify}(Q, \sigma)}{\mathsf{simplify}(Q, \forall Q'.\,\sigma) \doteq (\forall Q'.\,\sigma', \mathcal{L})}$$

**Figure 11.** Simplification

## 6. Adding predicates

With all the evidence machinery in place, it is now easy to add evidence translation for qualified types. Since we store predicates as part of the prefix, we only need to extend the definitions of type translation $(\_)^*$, evidence application app, and evidence abstraction abs as shown in Figure 10. We assume here that each language of predicates comes with a suitable translation function from predicates $\pi$ to runtime evidence of type $\pi^*$.

Picking up the examples from Section 4.1, a type class $C$ could be represented by a proper runtime dictionary of type $C^*$ that contains the member functions of $C$. An implicit argument predicate $(?x :: \sigma)$ is simply represented by a function of the same type: $(?x :: \sigma)^* = \sigma^*$. As a final example, a *has* predicate $(l :: \alpha \in r)$ for records could be represented by the runtime offset of $l$ in $r$, in other words, $(l :: \alpha \in r)^* = Int$.

There is one other place where we have to change the unification algorithm. In the (update) function (Figure 6), the abstraction check verifies if the new bound is polymorphic enough. The abstraction check algorithm needs to take predicates into account, namely, it must test if the two type schemes contain exactly the same predicates. Fortunately, predicates come with an entailment relation ($\Vdash$) that make this easy to verify. Since the type schemes in the abstraction check are already in an instance relation, we can consider two predicate sets equal when each predicate set entails the other.

Normally, an implementation of qualified types performs *simplification* where constant predicates are resolved to known evidence. For example, once a predicate $Num\ \alpha$ is instantiated to $Num\ Int$, we can eliminate the predicate and supply constant evidence at runtime. We assume that the language of predicates comes with an entailment relation that derives the evidence $e$ of a predicate $\pi$ under a prefix $Q$:

$$Q \Vdash \pi \rightsquigarrow e$$

For example, the expression $Q \Vdash Num\ Int \rightsquigarrow num_{Int}$ asserts that we can derive evidence $num_{Int}$ that $Num\ Int$ holds under some prefix $Q$. The derived evidence is in this case the runtime dictionary of the $Num$ class for $Int$. Of course, the entailment relation should only derive well-typed evidence terms:

$$Q \Vdash \pi \rightsquigarrow e \Rightarrow e :: \pi^*$$

Since the entailment works under a prefix instead of a set of predicates, this also allows for a satisfactory treatment of improvement [4], but a full discussion is beyond the scope of this paper.

Using the entailment relation, we can define a simplification algorithm for types, as shown in Figure 11. The expression $\mathsf{simplify}(Q, \sigma)$ simplifies $\sigma$ under prefix $Q$. It returns a pair of a simplified type $\sigma'$ and an evidence substitution $\mathcal{L}$. The evidence substitution will bind the evidence of predicates that are resolved during simplification. We assume that the argument to simplify is in normal form. The first two cases deal with mono types and $\bot$. The next case uses the entailment relation to simplify a resolved predicate. The evidence substitution is extended with the derived evidence for the predicate $\pi$. Since we work on normal forms, the final case simply ignores a bound and leaves the type as it is.

The type inference algorithm is now extended with the rule (simplify) that can be applied at any time to simplify the type of an expression.

$$(\text{simplify})\ \frac{\begin{array}{c} Q \mid \Gamma \vdash e : (Q', \sigma_1) \rightsquigarrow e^* \\ (\sigma_2, \mathcal{L}) = \mathsf{simplify}(Q', \mathsf{nf}(\sigma_1)) \\ e_2^* = \mathsf{abs}(\sigma_2)[\mathcal{L}\ \mathbf{in}\ \mathsf{app}(\sigma_1)[e^*]] \end{array}}{Q \mid \Gamma \vdash e : (Q', \sigma_2) \rightsquigarrow e_2^*}$$

We use the abstraction and application algorithm to construct a well-typed runtime term with the simplified type, where the evidence substitution resulting from simplification binds resolved predicates. Since we call simplify with the normalized form of $\sigma_1$, simplification can also speed up type inference without predicates as the abstraction and application functions can deal with simpler types.

## 7. Conclusion

In this article, we have shown how to combine the MLF type system with qualified types, and given an evidence relation to System F that demonstrates how to implement such a system efficiently. MLF with its impredicative first-class polymorphism has a number of advantages over comparable systems, and it seems so far that it scales well to programming languages with other type system extensions. In the future, we plan to analyze interactions of MLF with other features such as existential types.

We have given examples of MLF usage and demonstrated that it is convenient to use in practice, and that the added complexity can be kept hidden from the user most of the time. We would like to see MLF more widely used. In particular, we believe that MLF with qualified types can be seriously considered as an underlying type system for the Haskell programming language.

***Acknowledgements*** We are indebted to Didier Le Botlan for his extensive and constructive comments on a draft of this paper.

## References

[1] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Dept. of Computer Science, University of Nottingham, 1996.

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

[3] M. P. Jones. A theory of qualified types. In *4th. European Symposium on Programming (ESOP'92)*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306. Springer-Verlag, Feb. 1992.

[4] M. P. Jones. Simplifying and improving qualified types. Technical Report YALEU/DCS/RR-1040, Dept. of Computer Science, Yale University, 1994.

[5] D. Le Botlan. *ML$^F$: Une extension de ML avec polymorphisme de second ordre et instanciation implicite*. PhD thesis, INRIA Rocquencourt, May 2004.

[6] D. Le Botlan and D. Rémy. ML$^F$: raising ML to the power of system F. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 27–38. ACM Press, 2003.

[7] D. Leijen. Morrow: a row-oriented programming language. `http://www.cs.uu.nl/~daan/morrow.html`, July 2004.

[8] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[9] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[10] S. Peyton-Jones and M. Shields. Practical type inference for arbitrary-rank types. Submitted to the Journal of Functional Programming (JFP), 2004.

[11] C. Shan. Sexy types in action. *ACM SIGPLAN Notices*, 39(5):15–22, 2004.

## A. Supplemental algorithms

Useful domain:

$$\mathsf{dom}(Q \mathbin{/} \sigma) \doteq \mathsf{dom}(Q \mathbin{/} \mathsf{ftv}(\sigma))$$
$$\alpha \in \mathsf{dom}(Q \mathbin{/} \beta_1 \ldots \beta_n) \Leftrightarrow Q = (Q_1, \alpha \diamond \sigma, Q_2) \wedge \alpha \in \mathsf{ftv}(\forall Q_2.\, \beta_1 \to \cdots \to \beta_n \to ())$$

Constructed forms:

$$\begin{aligned}
&\mathsf{cf}(\tau) &&\doteq \tau \\
&\mathsf{cf}(\bot) &&\doteq \bot \\
&\mathsf{cf}(\forall(\alpha \diamond \sigma_\alpha).\, \sigma) &&\doteq \mathsf{cf}(\sigma_\alpha) \quad \text{if } \mathsf{nf}(\sigma) = \alpha \\
&\mathsf{cf}(\forall(\alpha \diamond \sigma_\alpha).\, \sigma) &&\doteq \forall(\alpha \diamond \sigma_\alpha).\, \mathsf{cf}(\sigma)
\end{aligned}$$

Free type variables:

$$\begin{aligned}
&\mathsf{ftv}(\alpha) &&\doteq \{\alpha\} \\
&\mathsf{ftv}(g\ \tau_1 \ldots \tau_n) &&\doteq \mathsf{ftv}(\tau_1) \cup \cdots \cup \mathsf{ftv}(\tau_n) \\
&\mathsf{ftv}(\bot) &&\doteq \varnothing \\
&\mathsf{ftv}(\forall(\alpha \diamond \sigma_\alpha).\, \sigma) &&\doteq (\mathsf{ftv}(\sigma) - \{\alpha\}) \cup \mathsf{ftv}(\sigma_\alpha) \quad \text{if } \alpha \in \mathsf{ftv}(\sigma) \\
&\mathsf{ftv}(\forall(\alpha \diamond \sigma_\alpha).\, \sigma) &&\doteq \mathsf{ftv}(\sigma) \quad \text{if } \alpha \notin \mathsf{ftv}(\sigma)
\end{aligned}$$

Splitting a prefix. The split algorithm takes a prefix $Q$ and a set of type variables $\overline{\alpha}$, and splits $Q$ in two parts $(Q_1, Q_2)$ such that the domain of $Q_1$ is the domain of $Q$ relevant to $\overline{\alpha}$.

$$() \uparrow \overline{\alpha} \doteq ((), ())$$
$$\frac{\alpha \in \overline{\alpha} \quad (Q_1, Q_2) = Q \uparrow (\overline{\alpha} - \alpha) \cup \mathsf{ftv}(\sigma)}{(Q, \alpha \diamond \sigma) \uparrow \overline{\alpha} \doteq ((Q_1, \alpha \diamond \sigma), Q_2)}$$
$$\frac{\alpha \notin \overline{\alpha} \quad (Q_1, Q_2) = Q \uparrow \overline{\alpha}}{(Q, \alpha \diamond \sigma) \uparrow \overline{\alpha} \doteq (Q_1, (Q_2, \alpha \diamond \sigma))}$$