# $\text{ML}^{\text{F}}$
# Raising ML to the Power of System F

Didier Le Botlan and Didier Rémy
INRIA-Rocquencourt
78153 Le Chesnay Cedex, France
{Didier.Le_Botlan,Didier.Remy}@inria.fr

## Abstract

*We propose a type system* $ML^F$ *that generalizes ML with first-class polymorphism as in System F. We perform partial type reconstruction. As in ML and in opposition to System F, each typable expression admits a principal type, which can be inferred. Furthermore, all expressions of ML are well-typed, with a possibly more general type than in ML, without any need for type annotation. Only arguments of functions that are used polymorphically must be annotated, which allows to type all expressions of System F as well.*

## 1 Introduction

### Type inference and first-class polymorphic types

Programming languages considerably benefit from static typechecking. In practice however, types may sometimes trammel programmers, from two opposite directions.

On the one hand, type annotations may quickly become a burden to write; while they usefully serve as documentation for toplevel functions, they also obfuscate the code when every local function must be decorated. On the other hand, since types are only approximations, any type system will reject programs that are perfectly well-behaved and that could be accepted by another more expressive one; hence, sharp programmers may be irritated in such situations.

Fortunately, solutions have been proposed to both of these problems: Type inference allows to elide most type annotations, which simultaneously relieves the programmer from writing such details and lightens programs. In parallel, more expressive type systems have been developed, so that programmers are less exposed to their limitations.

Unfortunately, those two situations are often conflicting. Expressive type systems tend to require an unbearable amount of type decorations, so that many of them only remained at the status of prototypes. Indeed, full type inference for System F is undecidable [Wel94]. Conversely, languages with simple type inference are still limited in expressiveness; more sophisticated type inference engines, such as those with subtyping constraints or higher-order unification have not yet been proved to work well in practice.

The ML language [DM82] appears to be a surprisingly stable point of equilibrium between those two forces: it combines a reasonably powerful yet simple type system and comes with an effective type inference engine. Besides, the ML experience made it clear that expressiveness of the type system and a significant amount of type inference are equally important.

Despite its success, ML could still be improved: indeed, there are real examples that require first-class polymorphic types; however infrequently these may occur, ML does not provide any reasonable alternative. (In fact, any tentative measure of the inconvenience can only be underestimated, since the lack of a full-fledged language to experiment with first-class polymorphism insidiously keeps programmers thinking in terms of toplevel polymorphism.) A first approach is to extend ML with first-class second-order polymorphism [LO94, Rém94, OL96, GR99]. However, the existing solutions are still limited in expressiveness and the amount of necessary type declarations keeps first-class polymorphism uneasy to use.

An alternative approach, initiated by Cardelli [Car93], is to start with an expressive but explicitly typed language, say $F^{\omega}_{:>}$, and perform a sufficient amount of type inference, so that simple programs—ideally including all ML programs—would not need any type annotation at all. This lead to *local type inference* [PT98], recently improved to *colored* local type inference [OZZ01]. These solutions are quite impressive. In particular, they include subtyping in combination with higher-order polymorphism. However, they fail to type all ML programs. Moreover, they also fail to provide an intuitive and simple specification of where type annotations are mandatory.

In this work, we follow the first approach. At least, by being conservative over ML, we are guaranteed to please programmers who are already quite happy with ML[1]. We build on some previous work [GR99], called Poly-ML, which has already been used to add polymorphic methods to OCaml [LDG+02]. Here, we retain the same expressiveness goal, that is, to type all System F programs, but eliminate the need to indicate coercions from polymorphic types to ML-types (the construct $\langle \cdot \rangle$ in Poly-ML). Our intention is also to provide for a simpler yet more flexible presentation of Poly-ML, on top of which further extensions, such as higher-order polymorphism—a feature that is highly desired to inherit from classes with polymorphic methods—could be built.

Actually, $\mathrm{ML}^{\mathrm{F}}$ is an alternative to explicitly-typed System F that can type the same set of untyped terms and potentially more, but with strictly fewer type information. Precisely, all type abstractions and type applications are left implicit, and type annotations on arguments of lambda abstractions are optional whenever the argument is used monomorphically in the body of the abstraction. Moreover, every term of $\mathrm{ML}^{\mathrm{F}}$ admits a principal type (which, in general, depends on its annotations).

The paper is organized as follows. In the rest of this section we will briefly recall the problems that arise when combining first-class polymorphic types with first-order-unification-based type inference, and introduce our solution, intuitively. In section 2, we will present our types. The core language is described in Section 3, including syntax, static and dynamic semantics. Section 4 presents some standard results, including type soundness and type inference. Section 5 introduces type annotations, which are mandatory to give the language its full power. In the last section (Section 6), we discuss language extensions, imperative features, and related works. For the sake of readability, some technical parts of the formal presentation, including the unification algorithms have been moved to the appendices. By lack of space all proofs are omitted.

## "Monomorphic abstraction of polymorphic types"

Explicit F-style polymorphism and implicit ML-style polymorphism are quite different in nature, and the two strategies enter in conflict almost immediately: in System F, the elimination of polymorphism is explicit, while in ML it is automatic at every occurrence of a polymorphic variable.

For illustration, let us combine two simple functions, namely apply the function `choose` defined as $\texttt{fun}\ (x)\ \texttt{fun}\ (y)\ \texttt{if}\ \mathit{true}\ \texttt{then}\ x\ \texttt{else}\ y$ to the identity function `id`. In ML, `choose` and `id` have principal types $\forall\,(\alpha)\ \alpha \to \alpha \to \alpha$ and $\forall\,(\alpha)\ \alpha \to \alpha$, respectively. For conciseness, we shall write $\texttt{id}_\alpha$ for $\alpha \to \alpha$. Should `choose id` have type $\sigma_1$ equal to $\forall\,(\alpha)\ \texttt{id}_\alpha \to \forall\,(\alpha)\ \texttt{id}_\alpha$, obtained by keeping the type of `id` uninstantiated, or have type $\sigma_2$ equal to $\forall\,(\alpha)\ (\texttt{id}_\alpha \to \texttt{id}_\alpha)$, obtained by instantiating the type of `id` to the monomorphic type $\texttt{id}_\alpha$ and generalizing $\alpha$ only at the end? Indeed, both $\sigma_1$ and $\sigma_2$ are correct types for `choose id`. However, neither one is more general than the other in System F. Indeed, the function `auto` defined as $\texttt{fun}\ (x : \forall\,(\gamma)\ \texttt{id}_\gamma)\ x\ x$, can be typed with $\sigma_1$, as `choose id`, but not with $\sigma_2$; otherwise `auto` could be applied, for instance, to the successor function, which would produce an error. Hence, $\sigma_1$ can not be safely coerced to $\sigma_2$. Conversely, however, there is a retyping function —a function whose type erasure $\eta$-reduces to the identity [Mit88]— from type $\sigma_2$ to type $\sigma_1$, that is, $\texttt{fun}\ (g : \sigma_2)\ \texttt{fun}\ (x : \forall\,(\alpha)\ \texttt{id}_\alpha)\ \texttt{fun}\ (\alpha)\ g\ \alpha\ (x\ \alpha)$. Actually, $\sigma_2$ is a principal type for `choose id` in $\mathrm{F}^{\eta*}$ (System F closed by $\eta$-expansion) [Mit88].

In fact, the argument of the function `choose id` does not have to be polymorphic: the function simply returns a value that is at best as polymorphic as both its argument and the identity. Conversely, the argument to the function `auto` must be at least as polymorphic as $\forall\,(\alpha)\ \texttt{id}_\alpha$. We could summarize these constraints by saying that:

$$\begin{array}{rl} \texttt{auto}: & \forall\,(\alpha = \forall\,(\gamma)\ \texttt{id}_\gamma)\ \alpha \to \alpha \\ \texttt{choose id}: & \forall\,(\alpha \geq \forall\,(\gamma)\ \texttt{id}_\gamma)\ \alpha \to \alpha \end{array}$$

Note that the type given to `choose id` captures the intuition that this application has type $\tau \to \tau$ for any instance $\tau$ of $\forall\,(\gamma)\ \texttt{id}_\gamma$. This form of quantification allows to postpone the decision of whether $\forall\,(\gamma)\ \texttt{id}_\gamma$ should be instantiated as soon as possible or kept polymorphic as long as possible. The bound of $\alpha$ can be weaken either by instantiating $\forall\,(\gamma)\ \texttt{id}_\gamma$ or by replacing $\geq$ by $=$. Hence, both `choose id succ` and `choose id auto` are well-typed, taking $\texttt{int} \to \texttt{int}$ or $\forall\,(\gamma)\ \texttt{id}_\gamma$ for the type of $\alpha$, respectively.

In fact, the type $\forall\,(\alpha \geq \forall\,(\gamma)\ \texttt{id}_\gamma)\ \alpha \to \alpha$ happens to be a principal type for `choose id` in $\mathrm{ML}^{\mathrm{F}}$. This type summarizes in a compact way the part of typechecking `choose id` that depends on the context in which it will be used: some typing constraints have been resolved definitely and forgotten; others, such that "$\alpha$ is any instance of $\forall\,(\gamma)\ \texttt{id}_\gamma$", are important and have been kept unresolved. In short, $\mathrm{ML}^{\mathrm{F}}$ provides richer types with constraints on the bounds of variables so that instantiation of these variables can be delayed until there is a principal way of instantiating them.

In our proposal, ML-style polymorphism, as in the type of `choose` or `id`, can be fully inferred. (We will show that all ML programs remain typable without type annotations.) Unsurprisingly, some polymorphic functions cannot be typed without annotations. For instance, $\texttt{fun}\ (x)\ x\ x$ cannot be typed in $\mathrm{ML}^{\mathrm{F}}$. In particular, we do not infer types for function arguments that are used polymorphically. Fortunately, such arguments can be annotated with a poly-

[1]On a practical level, this would also ensure upward compatibility of existing code, although translating tools could always be provided.

**Figure 1. Syntax of Types**

| | |
|---|---|
| $\tau ::= \alpha \mid g^n\ \tau_1\ ..\ \tau_n$ | Monotypes |
| $\sigma ::= \tau \mid \bot \mid \forall\,(\alpha \geq \sigma)\ \sigma \mid \forall\,(\alpha = \sigma)\ \sigma$ | Polytypes |

morphic type, as illustrated in the definition of `auto` given above. Once defined, a polymorphic function can be manipulated by another unannotated function, as long as the latter does not *use* polymorphism, which is then retained. This is what we qualify "*monomorphic abstraction of polymorphic types*". For instance, `id auto` (or `choose id auto`) remains of type $\forall\,(\alpha = \forall\,(\gamma)\ \mathtt{id}_\gamma)\ \alpha \to \alpha$, while `choose` and `id` do not have any type annotation. Finally, polymorphic functions can be used by implicit instantiation, much as in ML.

It should be noted that our notion of principal types is relative to partially annotated source terms, *i.e.* there is a type that captures all possible types of any given source term, but this type may depend on the type annotations that are present. For instance, both `fun` $(x : \forall\,(\alpha)\ \mathtt{id}_\alpha)\ x\ x$ and `fun` $(x : \forall\,(\alpha)\ \alpha)\ x\ x$ are typable in ML$^\mathsf{F}$, with incomparable types. (In our sense, explicitly typed System F, whose terms have unique types, admits principal types as well but implicitly typed System F does not.)

## 2 Types

### 2.1 Syntax of types

The syntax of types is given in Figure 1. The syntax is parameterized by an enumerable set of type variables $\alpha \in \vartheta$ and a family of type symbols $g \in \mathcal{G}$ given with their arity. To avoid degenerated cases, we assume that $\mathcal{G}$ contains at least a symbol of arity two (the infix arrow $\to$) and a symbol of arity zero (*e.g.* `unit`). We write $g^n$ if $g$ is of arity $n$. We also write $\bar{\tau}$ for tuples of types.

We distinguish between *monotypes*, and *polytypes*. By default, types refer to the more general forms, *i.e.* polytypes. As in ML, monotypes do not contain quantifiers. Polytypes generalize ML type schemes. Actually, ML type schemes can be seen as polytypes of the form $\forall\,(\alpha_1 \geq \bot)\ \ldots \forall\,(\alpha_n \geq \bot)\ \tau$ with outer quantifiers. Inner quantifiers as in System F cannot be written directly inside monotypes. However, they can be simulated with types of the form $\forall\,(\alpha = \sigma)\ \sigma'$, which stands, intuitively, for the polytype $\sigma'$ where all occurrences of $\alpha$ would have been replaced by the polytype $\sigma$ (our notation contains additional meaningful sharing information). Finally, the general form $\forall\,(\alpha \geq \sigma)\ \sigma'$ intuitively stands for the collection of *all polytypes $\sigma'$ where $\alpha$ is an instance of $\sigma$.*

**Notation** We say that $\alpha$ has a *rigid bound* in $(\alpha = \sigma)$ and a *flexible bound* in $(\alpha \geq \sigma)$. A particular case of flexible bound is the *unconstrained bound* $(\alpha \geq \bot)$, which we abbreviate as $(\alpha)$. For convenience, we write $(\alpha \diamond \sigma)$ for either $(\alpha = \sigma)$ or $(\alpha \geq \sigma)$. This acts as a meta-variable and two occurrences of $\diamond$ in the same context mean that they all stand for $=$ or all stand for $\geq$. To allow independent choices we use indices $\diamond_1$ and $\diamond_2$ for unrelated occurrences.

**Conversion** Type schemes are considered modulo $\alpha$-conversion where $\forall\,(\alpha\diamond\sigma)\ \sigma'$ binds $\alpha$ in $\sigma'$, but not in $\sigma$. We write $\mathsf{ftv}(\sigma)$ the set of free type variables of $\sigma$. Occurrences and free type variables are defined formally in Appendix A.

**Example 1** Quantifiers may only be outermost, as in ML, or in the bound of other bindings. Therefore, the type $\forall\,\alpha \cdot (\forall\,\beta \cdot (\tau[\beta] \to \alpha)) \to \alpha$ of System F[2] cannot be written directly. (Here, $\tau[\beta]$ means a type $\tau$ in which the variable $\beta$ occurs.) However, it could be represented by the type $\forall\,(\alpha)\ \forall\,(\beta' = \forall\,(\beta)\ \tau[\beta] \to \alpha)\ \beta' \to \alpha$. In fact, all types of System F can easily be represented as polytypes by recursively binding all occurrences of inner polymorphic types to fresh variables beforehand—see Appendix E for details. In this translation auxiliary variables are used in a linear way. For instance, $(\forall\,\alpha \cdot \tau) \to (\forall\,\alpha \cdot \tau)$ will be translated into $\forall\,(\alpha' = \forall\,(\alpha)\ \tau)\ \forall\,(\alpha'' = \forall\,(\alpha)\ \tau)\ \alpha' \to \alpha''$. Intuitively, $\forall\,(\alpha' = \forall\,(\alpha)\ \tau)\ \alpha' \to \alpha'$ could also represent the same System F type, more concisely. However, this type is not equivalent to the previous one, but only an instance of it, because it "shares more", as explained below.

Remark that type application is not possible in our system. Instead of explicit type application, as in System F, second-order types are instantiated implicitly as in ML along an instance relation, but a more elaborated one.

### 2.2 Type equivalence

The order of quantifiers and some other syntactical notations are not always meaningful. Such syntactic artifacts are captured by a notion of type equivalence. Type equivalence is relative to a prefix that specifies the bounds of free type variables.

A *prefix*, written $Q$, is a sequence of bindings, $(\alpha_1\ \diamond_1\ \sigma_1) \ldots (\alpha_n\ \diamond_n\ \sigma_n)$ where variables $\alpha_1, \ldots \alpha_n$ are pairwise distinct. We write $\mathsf{dom}(Q)$ for the set $\{\alpha_1\ \ldots \alpha_n\}$. We also write $\forall\,(Q)\ \sigma$ for $\forall\,(\alpha_1 \diamond_1 \sigma_1)\ \ldots (\alpha_n \diamond_n \sigma_n)\ \sigma$. Variables $\alpha_1 \ldots \alpha_n$ are not $\alpha$-convertible in the stand-alone prefix $Q$ (but they are in $\forall\,(Q)\ \sigma$). Since $\alpha_1, \ldots \alpha_n$ are pairwise distinct, we can unambiguously write $(\alpha \diamond \sigma) \in Q$ to mean that $Q$ is of the form $(Q_1, \alpha \diamond \sigma, Q_2)$.

[2] We write $\forall\,\alpha \cdot \tau$ for types of System F so as to avoid confusion.

**Figure 2. Type equivalence**

All rules are considered symmetrically.

Eq-Refl
$$(Q)\ \sigma \equiv \sigma$$

Eq-Trans
$$\frac{\begin{array}{c}(Q)\ \sigma_1 \equiv \sigma_2 \\ (Q)\ \sigma_2 \equiv \sigma_3\end{array}}{(Q)\ \sigma_1 \equiv \sigma_3}$$

Eq-Free
$$\frac{\alpha \notin \mathsf{ftv}(\sigma_1)}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1 \equiv \sigma_1}$$

Eq-Comm
$$\frac{\alpha_1 \notin \mathsf{ftv}(\sigma_2) \qquad \alpha_2 \notin \mathsf{ftv}(\sigma_1)}{(Q)\ \forall\,(\alpha_1 \diamond_1 \sigma_1)\ \forall\,(\alpha_2 \diamond_2 \sigma_2)\ \sigma \equiv \forall\,(\alpha_2 \diamond_2 \sigma_2)\ \forall\,(\alpha_1 \diamond_1 \sigma_1)\ \sigma}$$

Eq-Var
$$(Q)\ \forall\,(\alpha \diamond \sigma)\ \alpha \equiv \sigma$$

Eq-Context-R
$$\frac{(Q, \alpha \diamond \sigma)\ \sigma_1 \equiv \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1 \equiv \forall\,(\alpha \diamond \sigma)\ \sigma_2}$$

Eq-Context-L
$$\frac{(Q)\ \sigma_1 \equiv \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma_1)\ \sigma \equiv \forall\,(\alpha \diamond \sigma_2)\ \sigma}$$

Eq-Mono
$$\frac{(\alpha \diamond \sigma_0) \in Q \qquad (Q)\ \sigma_0 \equiv \tau_0}{(Q)\ \tau \equiv \tau[\tau_0/\alpha]}$$

The *equivalence under prefix* is a relation on triples composed of a prefix $Q$ and two types $\sigma_1$ and $\sigma_2$, written $(Q)\ \sigma_1 \equiv \sigma_2$. It is defined as the smallest relation that satisfies the rules of Figure 2. Rule Eq-Var says that a standalone variable is only an alias for its bound and may be replaced accordingly. Rule Eq-Mono means that binding of variables to monotypes can always be expanded, though it remains a convenient notation that allows for more natural and modular definitions. Rules Eq-Context-R and Eq-Context-L tell that $\equiv$ is a congruence; Rule Eq-Comm allows the reordering of independent binders; Rule Eq-Free eliminates unused bound variables.

Reasoning under prefixes allows to break up a type scheme $\forall\,(Q)\ \sigma$ and "look inside under prefix $Q$". For instance, it follows from iterations of rule Eq-Context-R that $(Q)\ \sigma \equiv \sigma'$ suffices to show $(\emptyset)\ \forall\,(Q)\ \sigma \equiv \forall\,(Q)\ \sigma'$.

In the examples, we often refer to the equivalence $(Q)\ \forall\,(\alpha \diamond \tau)\ \sigma \equiv \sigma[\tau/\alpha]$ as Eq-Mono$^\star$, which is provable by rules Eq-Mono, Eq-Context-R and Eq-Free.

**Notation** We write $\sigma_1 \equiv \sigma_2$ for $(\emptyset)\ \sigma_1 \equiv \sigma_2$. We write $\sigma \in \mathcal{T}$ if $\sigma \equiv \tau$ for some monotype $\tau$. We write $\sigma \in \mathcal{V}$ if $\sigma \equiv \alpha$ for some type variable $\alpha$.

## 2.3 Sharing instance

The equivalence under prefix is a symmetric operation (for a given prefix). In other words, it captures reversible transformations. Irreversible transformations are captured by instance relations. However, we distinguish between sharing instances, which can be reversed but only explicitly, and true instances, which are irreversible.

**Figure 3. Sharing instance**

Sh-Equiv
$$\frac{(Q)\ \sigma_1 \equiv \sigma_2}{(Q)\ \sigma_1 \sqsubseteq^{-} \sigma_2}$$

Sh-Trans
$$\frac{\begin{array}{c}(Q)\ \sigma_1 \sqsubseteq^{-} \sigma_2 \\ (Q)\ \sigma_2 \sqsubseteq^{-} \sigma_3\end{array}}{(Q)\ \sigma_1 \sqsubseteq^{-} \sigma_3}$$

Sh-Hyp
$$\frac{(\alpha_1 = \sigma_1) \in Q}{(Q)\ \sigma_1 \sqsubseteq^{-} \alpha_1}$$

Sh-Context-R
$$\frac{(Q, \alpha \diamond \sigma)\ \sigma_1 \sqsubseteq^{-} \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1 \sqsubseteq^{-} \forall\,(\alpha \diamond \sigma)\ \sigma_2}$$

Sh-Context-L
$$\frac{(Q)\ \sigma_1 \sqsubseteq^{-} \sigma_2}{(Q)\ \forall\,(\alpha = \sigma_1)\ \sigma \sqsubseteq^{-} \forall\,(\alpha = \sigma_2)\ \sigma}$$

The *sharing under prefix*, is a relation on triples composed of a prefix $Q$ and two types $\sigma_1$ and $\sigma_2$, written[3] $\sqsubseteq^{-}$, and defined as the smallest relation that satisfies the rules of Figure 3. Rules Sh-Context-L and Sh-Context-R are context rules; note that Rule Sh-Context-L does not allow sharing under flexible bounds. The interesting rule is Sh-Hyp, which shares a type $\sigma_1$ with a binding of the prefix. This operation can also be seen as *hiding the polymorphic type $\sigma_1$ under the abstract name $\alpha_1$.*

**Example 2** The judgement $(\alpha = \sigma')\ \forall\,(\alpha = \sigma')\ \sigma \sqsubseteq^{-} \sigma$ follows from Sh-Context-L (and other inessential rules). Unsharing can also act deeply: $\forall\,(\alpha = \forall\,(\alpha' = \sigma')\ \sigma)\ \sigma'' \sqsubseteq^{-} \forall\,(\alpha' = \sigma')\ \forall\,(\alpha = \sigma)\ \sigma''$ holds, provided $\alpha' \notin \mathsf{ftv}(\sigma'')$.

Remarkably, sharing two variables with the same bound is not reversible (sharing is not symmetric), unless the bound is equivalent to a monotype. In particular, when $(\alpha = \sigma) \in Q$, it is possible to derive $(Q)\ \sigma \sqsubseteq^{-} \alpha$ by rule Sh-Hyp, but $(Q)\ \alpha \sqsubseteq^{-} \sigma$ is not derivable (unless $\sigma \in \mathcal{T}$). This property is essential for type inference, since it prevents the inference of truly polymorphic types.

## 2.4 Type instance

The *instance under prefix* is a relation on triples composed of a prefix $Q$ and two types $\sigma_1$ and $\sigma_2$, written[4] $(Q)\ \sigma_1 \sqsubseteq \sigma_2$. It is defined as the smallest relation that satisfies the rules of Figure 4. We write $\sigma_1 \sqsubseteq \sigma_2$ for $(\emptyset)\ \sigma_1 \sqsubseteq \sigma_2$, as we do for equivalence.

[3]Read $\sigma_2$ is a sharing of $\sigma_1$, under prefix $Q$.

[4]Read $\sigma_2$ *is an instance of* $\sigma_1$—or $\sigma_1$ *is more general than* $\sigma_2$—*under prefix* $Q$.

**Figure 4. Type instance**

I-SHARE
$$\frac{(Q)\ \sigma_1 \mathrel{\sqsubset\!\!\!\!-} \sigma_2}{(Q)\ \sigma_1 \sqsubseteq \sigma_2}$$

I-TRANS
$$\frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2 \qquad (Q)\ \sigma_2 \sqsubseteq \sigma_3}{(Q)\ \sigma_1 \sqsubseteq \sigma_3}$$

I-BOT
$$(Q)\ \bot \sqsubseteq \sigma$$

I-CONTEXT-R
$$\frac{(Q, \alpha \diamond \sigma)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \forall\,(\alpha \diamond \sigma)\ \sigma_1 \sqsubseteq \forall\,(\alpha \diamond \sigma)\ \sigma_2}$$

I-CONTEXT-L
$$\frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \forall\,(\alpha \geq \sigma_1)\ \sigma \sqsubseteq \forall\,(\alpha \geq \sigma_2)\ \sigma}$$

I-HYP
$$\frac{(\alpha_1 \geq \sigma_1) \in Q}{(Q)\ \sigma_1 \sqsubseteq \alpha_1}$$

I-RIGID
$$\frac{}{(Q)\ \forall\,(\alpha \geq \sigma_1)\ \sigma \sqsubseteq \forall\,(\alpha = \sigma_1)\ \sigma}$$

Rule I-BOT means that $\bot$ behaves as the least element for the instance relation. Rules I-CONTEXT-L and I-RIGID mean that flexible bounds can be instantiated and changed into rigid bounds. Conversely, instantiation cannot occur under rigid bounds, except when restricted to the sharing relation $\mathrel{\sqsubset\!\!\!\!-}$ (as described by Rule SH-CONTEXT-L).

Rule I-HYP is the counter-part of rule SH-HYP. It can be used to share variables with identical bounds. For example, $(Q)\ \forall\,(\beta \geq \sigma)\ \forall\,(\alpha \geq \sigma)\ \alpha \to \beta \sqsubseteq \forall\,(\beta \geq \sigma)\ \forall\,(\alpha \geq \beta)\ \alpha \to \beta$ follows from rules I-CONTEXT-R, I-CONTEXT-L and I-HYP. Note that $\forall\,(\beta \geq \sigma)\ \forall\,(\alpha \geq \beta)\ \alpha \to \beta$ is equivalent to $\forall\,(\beta \geq \sigma)\ \beta \to \beta$ (by EQ-MONO$^\star$).

In the examples, we refer to the following derivable rule: $(Q)\ \forall\,(\alpha_1 \geq \forall\,(\alpha_2 \diamond \sigma_2)\ \sigma_1)\ \sigma \sqsubseteq \forall\,(\alpha_2 \diamond \sigma_2)\ \forall\,(\alpha_1 \geq \sigma_1)\ \sigma$ as I-UP, whenever $\alpha_2$ is not in $\mathsf{ftv}(\sigma)$.

**Example 3** As one would expect, the instance relation generalizes the instance relation of ML. For example, $\forall\,(\alpha)\ \tau[\alpha]$ is more general than $\forall\,(\alpha, \alpha')\ \tau[\alpha \to \alpha']$, as shown by the following derivation (valid under any prefix):

$$\begin{array}{cll}
 & \forall\,(\alpha)\ \tau[\alpha] & \\
= & \forall\,(\alpha \geq \bot)\ \tau[\alpha] & \text{by notation} \\
\equiv & \forall\,(\alpha', \alpha'')\ \forall\,(\alpha \geq \bot)\ \tau[\alpha] & \text{by EQ-FREE} \\
\sqsubseteq & \forall\,(\alpha', \alpha'')\ \forall\,(\alpha \geq \alpha' \to \alpha'')\ \tau[\alpha] & \text{by I-CONTEXT-L,} \\
 & & \text{since } (\alpha', \alpha'')\ \bot \sqsubseteq \alpha' \to \alpha'' \\
\equiv & \forall\,(\alpha', \alpha'')\ \tau[\alpha' \to \alpha''] & \text{by EQ-MONO}^\star \\
= & \forall\,(\alpha, \alpha')\ \tau[\alpha \to \alpha'] & \text{by renaming}
\end{array}$$

**Example 4** Conversely, $\forall\,(\alpha)\ \ \alpha$ is not an instance of $\forall\,(\alpha, \alpha')\ \alpha \to \alpha'$. Indeed, $\forall\,(\alpha)\ \alpha \sqsubseteq \forall\,(\alpha, \alpha')\ \alpha \to \alpha'$ is a particular case of Example 3. By way of contradiction, assume we have $\forall\,(\alpha, \alpha')\ \alpha \to \alpha' \sqsubseteq \forall\,(\alpha)\ \alpha$. By lemma 1, to be found below, we would have $\forall\,(\alpha, \alpha')\ \alpha \to \alpha' \equiv \forall\,(\alpha)\ \alpha$, which contradicts the fact that occurrences (defined in Appendix A) are preserved by equivalence, since $0$ is an occurrence of the left-hand side but not of the right-hand side.

**Example 5** The instance relation covers an interesting case of type isomorphism [Cos95]. In System F, type $\forall\,\alpha \cdot \tau \to \tau'$ is isomorphic[5] to $\tau \to \forall\,\alpha \cdot \tau'$ whenever $\alpha$ is not free in $\tau$. In ML$^{\mathrm{F}}$, the two corresponding type schemes are not equivalent but in an instance relation. Precisely, $\forall\,(\alpha' \geq \forall\,(\alpha)\ \tau')\ \tau \to \alpha'$ is more general than $\forall\,(\alpha)\ \tau \to \tau'$, as shown by the following derivation:

$$\begin{array}{ll}
\ \ \forall\,(\alpha' \geq \forall\,(\alpha)\ \tau')\ \tau \to \alpha' & \\
\sqsubseteq \forall\,(\alpha)\ \forall\,(\alpha' \geq \tau')\ \tau \to \alpha' & \text{by I-UP} \\
\equiv \forall\,(\alpha)\ \tau \to \tau' & \text{by EQ-MONO}^\star
\end{array}$$

As expected, the equivalence relation is the kernel of the instance relation.

**Lemma 1 (Equivalence)** *For all prefix $Q$ and types $\sigma$ and $\sigma'$, we have $(Q)\ \sigma \equiv \sigma'$ if and only if both $(Q)\ \sigma \sqsubseteq \sigma'$ and $(Q)\ \sigma' \sqsubseteq \sigma$ hold.*

**ML types**

Notice that ML-types can be seen as ML$^{\mathrm{F}}$ types where type schemes are restricted to have only unconstrained bounds. Then, the instance relation is the one of ML. In particular, $\forall\,(\bar{\alpha})\ \tau_0 \sqsubseteq \tau_1$ if and only if $\tau_1$ is of the form $\tau_0[\bar{\tau}/\bar{\alpha}]$.

## 2.5 Operation on prefixes

Rules SH-CONTEXT-L and I-CONTEXT-L show that two types $\forall\,(Q)\ \sigma$ and $\forall\,(Q')\ \sigma$ with the same suffix can be in an instance relation. Moreover, the suffix $\sigma$ does not matter (in these cases). This suggests a notion of inequality between prefixes alone. However, because prefixes are "open" this relation must be defined relatively to a set of variables that represents (a superset of) the free type variables of $\sigma$. In this context, a set of type variables is called an *interface* and is written $I$.

**Definition 1 (Prefix instance)** A prefix $Q$ is an *instance* of a prefix $Q'$ under the interface $I$, and we write $Q \sqsubseteq^I Q'$, if and only if $\forall\,(Q)\ \sigma \sqsubseteq \forall\,(Q')\ \sigma$ holds for all type $\sigma$ whose free variables are included in $I$. We omit $I$ in the notation when it is equal to $\mathsf{dom}(Q)$. We define $Q \equiv^I Q'$ and $Q \mathrel{\sqsubset\!\!\!\!-}^I Q'$ similarly. □

[5]That is, there exists a function $(\eta, \beta)$-reducible to the identity that transforms one into the other, and conversely.

**Figure 5. Expressions of** $\mathrm{ML^F}$

| | |
|---|---|
| $a ::= x \mid c \mid \texttt{fun}\ (x)\ a \mid a\ a \mid \texttt{let}\ x = a\ \texttt{in}\ a$ | Terms |
| $\mid (a : \star)$ | Guesspoints |
| $c ::= f \mid C$ | Constants |
| $z ::= x \mid c$ | Identifiers |

There exist equivalent inductive definitions for prefix instance, prefix sharing, and prefix equivalence, which we omit here by lack of space.

Prefixes can be seen as a generalization of the notion of substitutions to polytypes. Then, $Q \sqsubseteq Q'$ captures the usual notion of (a substitution) $Q$ being more general than (a substitution) $Q'$.

**Unification** The Appendix B defines an algorithm *unify* that given a prefix $Q$ and two types $\tau_1$ and $\tau_2$ returns a prefix that is the smallest instance of $Q$ that unifies $\tau_1$ and $\tau_2$ or fails if there is no instance of $Q$ that unifies $\tau_1$ and $\tau_2$.

**Lemma 2 (Soundness of unification)** *If unify* $(Q, \tau, \tau')$ *succeeds with* $Q'$*, then we have* $Q \sqsubseteq Q'$ *and* $(Q')\ \tau \equiv \tau'$.

**Lemma 3 (Completeness of unification)** *Given a prefix* $Q$ *and two types* $\tau$ *and* $\tau'$ *closed under* $Q$*, if there exists a prefix* $Q_b$ *such that* $Q \sqsubseteq^{\mathsf{dom}(Q)} Q_b$ *and* $(Q_b)\ \tau \equiv \tau'$*, then unify* $(Q, \tau, \tau')$ *succeeds with some prefix* $Q_a$ *such that* $Q_a \sqsubseteq^{\mathsf{dom}(Q)} Q_b$.

As a corollary, for any given interface $I$, the relation $\sqsubseteq^I$ defines an upper semi-lattice on the set of prefixes.

## 3 The core language

We formalize our approach as a small extension to core ML that is parameterized by a set of constants. Expressions of $\mathrm{ML^F}$ are those of ML. We also consider a variant of $\mathrm{ML^F}$, called $\mathrm{ML^{F\star}}$, in which some expressions have been annotated (intuitively, places to call an horacle during type inference). We represent annotations by an additional node $(a : \star)$, called a guesspoint. The syntax of expressions of $\mathrm{ML^{F\star}}$, written with letter $a$, is given in Figure 5 and expressions of $\mathrm{ML^F}$ are those that do not contain guesspoints (guesspoints may be introduced during reduction, but source programs, for which types will be inferred, should not contain them). We assume given a countable set of variables $x \in \mathcal{V}$ and a countable set of constants $c \in \mathcal{C}$. Every constant $c$ comes with its arity $|c|$. A constant is either a primitive $f$ or a constructor $C$. The distinction between constructors and primitives lies in their dynamic semantics. We use letter $z$ to refer to identifiers, *i.e.* either variables or constants.

### 3.1 Static semantics

Typing contexts, written with letter $\Gamma$ are lists of assertions of the form $z : \sigma$. We write $z : \sigma \in \Gamma$ to mean that $z$ is bound in $\Gamma$ and $z : \sigma$ is its rightmost binding in $\Gamma$. We assume given an initial context $\Gamma_0$ mapping constants to closed polytypes.

Typing judgments are of the form $(Q)\ \Gamma \vdash a : \sigma$. A small difference with ML is the presence of the prefix $Q$ that must assign bounds to type variables appearing free in $\Gamma$ or $\sigma$. By comparison, this prefix is left implicit in ML because free variables all have the same (implicit) bound $\perp$. On the contrary, we require that $\sigma$ and all type schemes of $\Gamma$ be closed with respect to $Q$, that is, $\mathsf{ftv}(\Gamma) \cup \mathsf{ftv}(\sigma) \subseteq \mathsf{dom}(Q)$.

**Typing rules** The typing rules of $\mathrm{ML^F}$ are described in Figure 6. They correspond to the typing rules of ML modulo the richer types, the richer instance relation, and the explicit binding of free variables in judgments. The language $\mathrm{ML^{F\star}}$ requires an additional typing rule GUESS. (This rule would have no effect in ML where sharing $\sqsubseteq$ would be the same as $\equiv$ in ML.)

As in ML, there is an important difference between rule FUN and rule LET: while typechecking their bodies, a let-bound variable can be assigned a type scheme, but a $\lambda$-bound variable can only be assigned a simple type in $\Gamma$. Indeed, the latter must be guessed while the former can be inferred from the type of the bound expression. This restriction is essential to enable type inference. Note that a $\lambda$-bound variable can be assigned a polytype indirectly, via a type variable $\alpha$ bound to a type scheme $\sigma$ in $Q$. However, this will not allow to take different instances of $\sigma$ while typing the body of the abstraction. Indeed, the only possible instances of $\alpha$ under a prefix $Q$ that contains the binding $(\alpha \diamond \sigma)$ are types $\sigma'$ that are equivalent to $\alpha$ under $Q$. However, $\alpha$ is not then equivalent to $\sigma$ under $Q$, even if the bound of $\alpha$ in $Q$ is rigid. Thus, if $x : \alpha$ is in the typing context $\Gamma$, the only way of typing $x$ (modulo equivalence) is $(Q)\ \Gamma \vdash x : \alpha$, whereas $(Q)\ \Gamma \vdash x : \sigma$ is not derivable.

**The language ML as a subset of $\mathrm{ML^F}$** ML can be embedded into $\mathrm{ML^F}$ by restricting type schemes to those of ML. Then, the prefix $Q$ only records the set of free type variables of the judgment. In particular, rules GEN and INST are then exactly those of ML. Hence any closed program typable in ML is also typable in $\mathrm{ML^F}$.

**Figure 6. Typing rules for** $\mathrm{ML}^{\mathrm{F}}$

$$\textsc{Var}\ \frac{z : \sigma \in \Gamma}{(Q)\ \Gamma \vdash z : \sigma} \qquad \textsc{Fun}\ \frac{(Q)\ \Gamma, x : \tau_0 \vdash a : \tau}{(Q)\ \Gamma \vdash \mathtt{fun}\ (x)\ a : \tau_0 \to \tau}$$

$$\textsc{App}\ \frac{(Q)\ \Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad (Q)\ \Gamma \vdash a_2 : \tau_2}{(Q)\ \Gamma \vdash a_1\ a_2 : \tau_1}$$

$$\textsc{Let}\ \frac{(Q)\ \Gamma \vdash a_1 : \sigma \qquad (Q)\ \Gamma, x : \sigma \vdash a_2 : \tau}{(Q)\ \Gamma \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau}$$

$$\textsc{Gen}\ \frac{(Q, \alpha \diamond \sigma)\ \Gamma \vdash a : \sigma' \qquad \alpha \notin \mathsf{ftv}(\Gamma)}{(Q)\ \Gamma \vdash a : \forall\,(\alpha \diamond \sigma)\ \sigma'}$$

$$\textsc{Inst}\ \frac{\begin{array}{c}(Q)\ \Gamma \vdash a : \sigma \\ (Q)\ \sigma \sqsubseteq \sigma'\end{array}}{(Q)\ \Gamma \vdash a : \sigma'} \qquad \textsc{Guess}\ \frac{\begin{array}{c}(Q)\ \Gamma \vdash a : \sigma \\ (Q)\ \sigma' \sqsubseteq\!\!\!\!\!= \sigma\end{array}}{(Q)\ \Gamma \vdash (a : \star) : \sigma'}$$

**Example 6** This first example of typing illustrates the role of polytype schemes in typing derivations: we consider the simple expression $K$ defined by $\mathtt{fun}\ (y)\ \mathtt{fun}\ (x)\ x$ and its typing derivation as would be done in ML (we recall that $(\alpha, \alpha')$ stands for $(\alpha \geq \bot, \alpha' \geq \bot)$):

$$\textsc{Gen}\ \dfrac{\textsc{Fun}\ \dfrac{\textsc{Fun}\ \dfrac{(\alpha, \alpha')\ y : \alpha, x : \alpha' \vdash x : \alpha'}{(\alpha, \alpha')\ y : \alpha \vdash \mathtt{fun}\ (x)\ x : \alpha' \to \alpha'}}{(\alpha, \alpha')\ \vdash K : \alpha \to (\alpha' \to \alpha')}}{\vdash K : \forall\,(\alpha, \alpha')\ \alpha \to (\alpha' \to \alpha')}$$

We show another typing derivation that actually infers a more general type for $K$ in $\mathrm{ML}^{\mathrm{F}}$. For conciseness we write $\sigma_{\mathtt{id}}$ for $\forall\,(\alpha')\ \alpha' \to \alpha'$ and $Q$ for $\alpha, \beta \geq \sigma_{\mathtt{id}}$.

$$\textsc{Gen}\ \dfrac{\textsc{Fun}\ \dfrac{\textsc{Inst}\ \dfrac{\begin{array}{c}\textsc{Gen}\ \dfrac{\textsc{Fun}\ \dfrac{(Q, \alpha')\ y : \alpha, x : \alpha' \vdash x : \alpha'}{(Q, \alpha')\ y : \alpha \vdash \mathtt{fun}\ (x)\ x : \alpha' \to \alpha'}}{(Q)\ y : \alpha \vdash \mathtt{fun}\ (x)\ x : \sigma_{\mathtt{id}}} \\ (Q)\ \sigma_{\mathtt{id}} \sqsubseteq \beta\end{array}}{(Q)\ y : \alpha \vdash \mathtt{fun}\ (x)\ x : \beta}}{(Q)\ \vdash K : \alpha \to \beta}}{\vdash K : \forall\,(Q)\ \alpha \to \beta}$$

In example 5, we have shown that $\forall\,(Q)\ \alpha \to \beta$ is more general than $\forall\,(\alpha, \alpha')\ \alpha \to (\alpha' \to \alpha')$.

**Example 7** This example illustrates how sharing in types controls "access to polymorphism", which is the key to having principal types and type inference. Let $f$ and $g$ be two functions of respective types $\sigma_1 \triangleq \forall\,(\alpha = \sigma_{\mathtt{id}})\ \forall\,(\alpha' = \sigma_{\mathtt{id}})\ \alpha \to \alpha'$ and $\sigma_2 \triangleq \forall\,(\alpha = \sigma_{\mathtt{id}})\ \alpha \to \alpha$. Then, the expression $\mathtt{fun}\ (x)\ f\ x\ x$ is typable but $\mathtt{fun}\ (x)\ g\ x\ x$ is not. In the former case, we can easily derive $(\alpha = \sigma_{\mathtt{id}}, \alpha' = \sigma_{\mathtt{id}})\ x : \alpha \vdash f\ x : \alpha'$ (1). Hence, by rule Gen, $(\alpha = \sigma_{\mathtt{id}})\ x : \alpha \vdash f\ x : \forall\,(\alpha' = \sigma_{\mathtt{id}})\ \alpha'$ since $\alpha'$ is not free in the context. We have $(\alpha = \sigma_{\mathtt{id}})\ \forall\,(\alpha' = \sigma_{\mathtt{id}})\ \alpha' \sqsubseteq \alpha \to \alpha$, since, by Rule Eq-Var we have $(\alpha = \sigma_{\mathtt{id}})\ \forall\,(\alpha' = \sigma_{\mathtt{id}})\ \alpha' \sqsubseteq \sigma_{\mathtt{id}}$ and $(\alpha = \sigma_{\mathtt{id}})\ \sigma_{\mathtt{id}} \sqsubseteq \alpha \to \alpha$. Thus, by Rule Inst, we get $(\alpha = \sigma_{\mathtt{id}})\ x : \alpha \vdash f\ x : \alpha \to \alpha$ and $\vdash \mathtt{fun}\ (x)\ f\ x\ x : \sigma_{\mathtt{id}}$ follows. However, when $f$ is replaced by $g$, we only get $(\alpha = \sigma_{\mathtt{id}})\ x : \alpha \vdash f\ x : \alpha$ instead of the judgment (1) and we cannot apply Rule Gen.

**Example 8** To pursue Example 7, let `choose` be a function of type $\forall\,(\alpha)\ \alpha \to \alpha \to \alpha$, which can be used to force unification of the types of its two arguments. The type of $\mathtt{choose}\ f\ \mathtt{id}$ is then $\sigma_2$, which is the least common instance of both $\sigma_1$ and $\sigma_{\mathtt{id}}$ (1). More precisely, we have the two following derivations where $\Gamma$ is $(\mathtt{choose} : \forall\,(\alpha)\ \alpha \to \alpha \to \alpha, f : \sigma_1; \mathtt{id} : \sigma_{\mathtt{id}})$ and $Q$ is $\alpha \geq \sigma_2$.

$$\textsc{Inst}\ \frac{\begin{array}{c}(Q)\ \Gamma \vdash f : \sigma_1 \\ (Q)\ \sigma_1 \sqsubseteq \alpha\ (2)\end{array}}{(Q)\ \Gamma \vdash f : \alpha\ (4)} \qquad \frac{\begin{array}{c}(Q)\ \Gamma \vdash \mathtt{id} : \sigma_{\mathtt{id}} \\ (Q)\ \sigma_{\mathtt{id}} \sqsubseteq \alpha\ (3)\end{array}}{(Q)\ \Gamma \vdash \mathtt{id} : \alpha\ (5)}\ \textsc{Inst}$$

Indeed, (2) and (3) follows from $\sigma_1 \sqsubseteq \sigma_2$ and $\sigma_{\mathtt{id}} \sqsubseteq \sigma_2$ by (1) and Rule I-Hyp. Then, we can easily derive $\Gamma \vdash \mathtt{choose}\ f\ \mathtt{id} : \sigma_2$.

## 3.2 Syntax directed presentation

As in ML, we can replace the typing rules by a set of equivalent but syntax-directed rules, which are given in Figure 7. Here however, Rule Gen would also need to be applied before Rule Fun and the right-hand side of Rule App. Instead, we keep typing judgments generalized (*vs.* instantiated in ML) as much as possible.

**Example 9** As we claimed in the introduction, a variable that is used polymorphically in the body of an abstraction must be annotated. Let us check that $\mathtt{fun}\ (x)\ x\ x$ is not typable in $\mathrm{ML}^{\mathrm{F}}$. By contradiction, a syntax-directed type derivation of this expression would be of the form:

$$\textsc{Fun}^{\triangle}\ \dfrac{\textsc{App}^{\triangle}\ \dfrac{\begin{array}{cc}\textsc{Var}^{\triangle}\ (Q)\ x : \tau_0 \overset{\triangle}{\vdash} x : \tau_0 & (Q)\ x : \tau_0 \overset{\triangle}{\vdash} x : \tau_0\ \textsc{Var}^{\triangle} \\ (Q)\ \tau_0 \sqsubseteq \tau_2 \to \tau_1\ (1) & (Q)\ \tau_0 \sqsubseteq \tau_2\ (2)\end{array}}{(Q)\ x : \tau_0 \overset{\triangle}{\vdash} x\ x : \tau_1}}{(Q)\ \emptyset \vdash \mathtt{fun}\ (x)\ x\ x : \tau_0 \to \tau_1}$$

**Figure 7. Syntax directed typing rules**

$$\mathrm{VAR}^{\triangle}\quad \dfrac{z : \sigma \in \Gamma}{(Q)\ \Gamma \overset{\triangle}{\vdash} z : \sigma}$$

$$\mathrm{FUN}^{\triangle}\quad \dfrac{(QQ')\ \Gamma, x : \tau_0 \overset{\triangle}{\vdash} a : \sigma \qquad \mathsf{dom}(Q') \cap \Gamma = \emptyset}{(Q)\ \Gamma \overset{\triangle}{\vdash} \mathtt{fun}\ (x)\ a : \forall\,(Q', \alpha \geq \sigma)\ \tau_0 \to \alpha}$$

$$\mathrm{APP}^{\triangle}\quad \dfrac{\begin{array}{cc}(Q)\ \Gamma \overset{\triangle}{\vdash} a_1 : \sigma_1 & (Q)\ \Gamma \overset{\triangle}{\vdash} a_2 : \sigma_2 \\ (Q)\ \sigma_1 \sqsubseteq \forall\,(Q')\ \tau_2 \to \tau_1 & (Q)\ \sigma_2 \sqsubseteq \forall\,(Q')\ \tau_2\end{array}}{(Q)\ \Gamma \overset{\triangle}{\vdash} a_1\ a_2 : \forall\,(Q')\ \tau_1}$$

$$\mathrm{LET}^{\triangle}\quad \dfrac{(Q)\ \Gamma \overset{\triangle}{\vdash} a_1 : \sigma_1 \qquad (Q)\ \Gamma, x : \sigma_1 \overset{\triangle}{\vdash} a_2 : \sigma_2}{(Q)\ \Gamma \overset{\triangle}{\vdash} \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \sigma_2}$$

$$\mathrm{GUESS}^{\triangle}\quad \dfrac{(Q)\ \Gamma \overset{\triangle}{\vdash} a : \sigma \qquad (Q)\ \sigma \sqsubseteq \sigma'' \qquad (Q)\ \sigma' \mathrel{-\!\!\!\!\sqsubseteq} \sigma''}{(Q)\ \Gamma \overset{\triangle}{\vdash} (a : \star) : \sigma'}$$

Hence, (1) and (2) must hold simultaneously, which implies $(Q)\ \tau_2 \equiv \tau_2 \to \tau_1$ (we omit the proof by lack of space). However, this contradicts Lemma 9 to be found in Appendix A.

This example shows the limit of type inference, which is actually the strength of our system! *i.e.* to maintain principal types by rejecting examples where type inference would need to "guess".

**Example 10** Let us recover typability by introducing a guesspoint in $\mathtt{fun}\ (x)\ (x : \star)\ x$. Taking $(\alpha = \sigma_{\mathtt{id}})$ for $Q$ and $\alpha$ for $\tau_0$, we obtain the following derivation:

$$\mathrm{FUN}^{\triangle}\ \dfrac{\mathrm{APP}^{\triangle}\ \dfrac{\mathrm{GUESS}^{\triangle}\ \dfrac{\begin{array}{c}\mathrm{VAR}^{\triangle}\ (Q)\ x : \alpha \overset{\triangle}{\vdash} x : \alpha \\ (Q)\ \sigma_{\mathtt{id}} \mathrel{-\!\!\!\!\sqsubseteq} \alpha\ (1)\end{array}}{(Q)\ x : \alpha \overset{\triangle}{\vdash} (x : \star) : \sigma_{\mathtt{id}}} \qquad \begin{array}{c}(Q)\ \sigma_{\mathtt{id}} \sqsubseteq \alpha \to \alpha \\ (Q)\ x : \alpha \overset{\triangle}{\vdash} x : \alpha\ \mathrm{VAR}^{\triangle}\end{array}}{(Q)\ x : \alpha \overset{\triangle}{\vdash} (x : \star)\ x : \alpha}}{\vdash \mathtt{fun}\ (x)\ (x : \star)\ x : \forall\,(\alpha = \sigma_{\mathtt{id}})\ \alpha \to \alpha}$$

The crucial role of the annotation is to allow the use of (1) to enrich the typing of $x$: while (1) implies $(Q)\ \sigma_{\mathtt{id}} \sqsubseteq \alpha$, this relation cannot be inverted, so rule INST cannot be used to replace $\alpha$ by its bound $\sigma_{\mathtt{id}}$. The guesspoint allows such going backward, but only along the $\mathrel{-\!\!\!\!\sqsubseteq}$ relation—replacing $\mathrel{-\!\!\!\!\sqsubseteq}$ by $\sqsubseteq$ in rule GUESS would be unsound.

Section 5 shows how type annotations can be used to infer a type for $\mathtt{fun}\ (x : \sigma)\ x\ x$ when $\sigma$ is given explicitly.

## 3.3 Dynamic semantics

The semantics of $\mathrm{ML}^{\mathrm{F}\star}$ is the standard call-by-value semantics of ML. We present it as a small-step reduction semantics. Values and call-by-value evaluation contexts are described below.

$$\begin{array}{rll}
v ::= & w \mid (w : \star) & \\
w ::= & \mathtt{fun}\ (x)\ a & \\
 & \mid f\ v_1\ \ldots v_n & n < |f| \\
 & \mid C\ v_1\ \ldots v_n & n \leq |C| \\
E ::= & [\,] \mid E\ a \mid v\ E \mid (E : \star) \mid \mathtt{let}\ x = E\ \mathtt{in}\ a &
\end{array}$$

The reduction relation $\longrightarrow$ is parameterized by a set of $\delta$-rules of the form:

$$f\ v_1\ \ldots v_n \longrightarrow a \qquad \text{when } |f| = n \qquad (\delta)$$

Constants $c$ are given with their (closed) types by an initial typing environment $\Gamma_0$.

$$\begin{array}{rcll}
(\mathtt{fun}\ (x)\ a)\ v & \longrightarrow & v[a/x] & (\beta_v) \\
\mathtt{let}\ x = v\ \mathtt{in}\ a & \longrightarrow & v[a/x] & (\beta_{let}) \\
(v_1 : \star)\ v_2 & \longrightarrow & (v_1\ v_2 : \star) & (\star) \\
((v : \star) : \star) & \longrightarrow & (v : \star) & (\star\star)
\end{array}$$

The main reduction is the $\beta$-reduction rule that takes two forms $\beta_v$ and $\beta_{let}$. The two $\star$-rules deal with guesspoints. Guesspoints are maintained during the reduction to which they do not contribute. They are simply pushed out of applications and collapse when they meet.

Finally, the reduction is the smallest relation containing $\delta$, $\beta_v$, $\beta_{let}$, and the two $\star$-rules and closed by congruence:

$$E[a] \longrightarrow E[a'] \text{ if } a \longrightarrow a' \qquad (\mathrm{CONTEXT})$$

## 4 Formal properties

We state a few standard properties of $\mathrm{ML}^{\mathrm{F}}$ in appendix C. Below, we verify type soundness and we address type inference. By lack of space, all proofs are omitted.

### 4.1 Type soundness

As usual, type soundness is shown by a combination of *subject reduction*, which ensures that typings are preserved by reduction, and *progress*, which ensures that well-typed programs that are not values can be further reduced.

To ease the presentation, we introduce a relation $\subseteq$ between programs: we write $a \subseteq a'$ if and only if every typing of $a$, *i.e.* a triple $(Q, \Gamma, \sigma)$ such that $(Q)\ \Gamma \vdash a : \sigma$ holds, is also a typing of $a'$. A relation $\mathcal{R}$ on programs preserves typings whenever it is a sub-relation of $\subseteq$.

Of course type soundness cannot hold without some assumptions on the semantics of constants.

**Hypotheses** We assume that the following three properties hold for constants.

**(H0) (Arity)** Each constant $c \in \mathsf{dom}(\Gamma_0)$ has a closed type $\Gamma_0(c)$ of the form $\forall\,(Q)\;\tau_1 \to \ldots \tau_{|c|} \to \tau$ and such that the top symbol of $\tau$ (*i.e.* $(\forall\,(Q)\;\tau)/\epsilon$ as defined in the appendix A) is not in $\{\to, \bot\}$ whenever $c$ is a constructor.

**(H1) (Subject-Reduction)** All $\delta$-rules preserve typings.

**(H2) (Progress)** Any expression $a$ of the form $f\;v_1 \ldots v_{|f|}$, such that $(Q)\;\Gamma_0 \vdash a : \sigma$ is in the domain of $(\delta)$

**Theorem 1 (Subject reduction)** *Reduction preserves typings.*

**Theorem 2 (Progress)** *Any expression $a$ such that $(Q)$ $\Gamma_0 \vdash a : \sigma$ is a value or can be further reduced.*

## 4.2 Type inference

Type inference is similar to type inference for ML: it follows the syntax-directed typing rules and reduces to (prefix) unification.

Namely, a *type inference problem* is a triple $(Q, \Gamma, a)$, where all free type variables in $\Gamma$ are bound in $Q$. A pair $(Q', \sigma)$ is a *solution* to this problem if $Q \sqsubseteq Q'$ and $(Q')$ $\Gamma \vdash a : \sigma$. A pair $(Q', \sigma')$ is an *instance* of a pair $(Q, \sigma)$ if $Q \sqsubseteq Q'$ and $(Q')\;\sigma \sqsubseteq \sigma'$. A solution of a type inference problem is *principal* if all other solutions are instances of the given one. Figure 9 in the Appendix D defines a type inference algorithm for ML$^{\mathrm{F}}$.

**Theorem 3 (Type inference)** *The set of solutions of a solvable type inference problem admits a principal solution. Given any type inference problem, the algorithm W$^{\mathrm{F}}$ either returns a principal solution or fails if no solution exists.*

## 5 Type annotations

Although ML$^{\mathrm{F}}$ has richer types than ML, nude-ML$^{\mathrm{F}}$, *i.e.* ML$^{\mathrm{F}}$ without constants cannot type more programs than ML. This is stated precisely by the following lemma (the inverse inclusion has already been stated in Section 3.1):

**Lemma 4** *If the judgment $(Q)\;\Gamma \vdash a : \sigma$ holds in ML$^{\mathrm{F}}$ where the typing context $\Gamma$ contains only ML types and $Q$ contains only type variables with unconstrained bounds, then there exists a derivation of $\Gamma \vdash a : \forall\,(\bar\alpha)\;\tau$ in ML where $\forall\,(\bar\alpha)\;\tau$ is obtained from $\sigma$ by moving all inner quantifiers ahead.*

In particular, closed expressions without constants that can be typed in ML$^{\mathrm{F}}$ can always be typed in the empty environment and under the empty prefix, and therefore can also be typed in ML.

This is not the case with ML$^{\mathrm{F}\star}$, in which the expression `fun` $(x)$ $(x : \star)$ $x$ is typable. As we shall see below all terms of System F can be typed in ML$^{\mathrm{F}\star}$. Fortunately, there is an interesting wrapping around nude-ML$^{\mathrm{F}}$ that provides the same expressiveness as ML$^{\mathrm{F}\star}$ while retaining type inference. Precisely, we provide type annotations as a collection of coercion primitives, *i.e.* functions that change the type of expressions without changing their meaning. The following example, which describes the case of a single annotation, should provide intuition for the general case.

**Example 11** Let $\sigma_{\mathtt{id}}$ be the type of the identity function $\forall\,(\alpha)\;\alpha \to \alpha$ and assume given a constant `id` of type $\forall\,(\alpha = \sigma_{\mathtt{id}}, \alpha' \geq \sigma_{\mathtt{id}})\;\alpha \to \alpha'$ with the $\delta$-reduction `id` $v \longrightarrow (v : \star)$. Then, the expression $a$ defined as `fun` $(x)$ `let` $x$ = `id` $x$ `in` $x$ $x$ is well-typed (see Example 7), also of type $\forall\,(\alpha = \sigma_{\mathtt{id}}, \alpha' \geq \sigma_{\mathtt{id}})\;\alpha \to \alpha'$. The effect of `id` is to remove sharing between the type of the argument of the coercion, and the type returned by the coercion. Of course, since the connection between the argument and the result is lost, we must force the argument to be at least as polymorphic as the return type, hence the use of a rigid bound on the left. Indeed, it would not be correct to apply `id` to a value of type `int` $\to$ `int`, since the result would not have type $\sigma_{\mathtt{id}}$.

### 5.1 Annotation primitives

We call *annotations* the following denumerable collection of primitives:

$$(\exists\,(Q)\;\sigma) : \forall\,(Q)\;\forall\,(\alpha = \sigma)\;\forall\,(\beta = \sigma)\;\alpha \to \beta \qquad \in \Gamma_0$$

for all prefixes $Q$ and type schemes $\sigma$ such that $\forall\,(Q)\;\sigma$ is closed. We identify annotation primitives up to the equivalence of their type. We write $(a : \exists\,(Q)\;\sigma)$ for $(\exists\,(Q)\;\sigma)\;a$. We also abbreviate $(\exists\,(Q)\;\sigma)$ as $(\sigma)$ when all bounds in $Q$ are unconstrained.

While annotations are introduced as primitives, for simplicity of presentation, they are meant to be applied. The type of the annotation primitive may be instantiated, but the unsharing effect of the annotation still applies. This is described by the following technical lemma, which has similarities with the Rule ANNOT of Poly-ML [GR99].

**Lemma 5** *The judgment $(Q_0)\;\Gamma \vdash (a : \exists\,(Q)\;\sigma) : \sigma_0$ is valid if and only if there exist two types $\forall\,(Q')\;\sigma_1'$ and $\forall\,(Q')\;\sigma_0'$ such that the judgement $(Q_0)\;\Gamma \vdash a : \forall\,(Q')\;\sigma_1'$ holds together with the following relations:*

$$Q_0 Q \sqsubseteq Q_0 Q' \qquad (Q_0 Q')\;\sigma \sqsupseteq\!\!\!\!\sqsubseteq \sigma_1'$$

$$(Q_0 Q')\;\sigma \sqsupseteq\!\!\!\!\sqsubseteq \sigma_0' \qquad (Q_0)\;\forall\,(Q')\;\sigma_0' \sqsubseteq \sigma_0$$

**Corollary 6** *The judgment* $(Q_0)\ \Gamma \vdash (a : \star) : \sigma_0$ *holds if and only if there exists an annotation* $(\exists\,(Q)\ \sigma)$ *such that* $(Q_0)\ \Gamma \vdash (a : \exists\,(Q)\ \sigma) : \sigma_0$ *holds.*

**Corollary 7** *A term is typable in ML*$^{F}$ *if and only if it is typable in ML*$^{F^\star}$ *after replacement of all applications of annotations by guesspoints around their arguments.*

**Reduction of annotations** The $\delta$-reduction for annotations just replaces explicit type information by guesspoints.

$$(v : \exists\,(Q)\ \sigma) \longrightarrow (v : \star)$$

**Lemma 8 (Soundness of type annotations)** *The three hypotheses (H0, arity), (H1, subject-reduction), and (H2, progress) hold when primitives are the set of simple annotations, alone.*

**Syntactic sugar** In practice, annotations are rather put on abstractions than on use sites. Indeed, the annotation should then work for all use sites. Annotations on abstractions `fun` $(x : \sigma)\ a$ can be seen as syntactic sugar for `fun` $(x)$ `let` $x = (x : \sigma)$ `in` $a$. The derived typing rule is:

FUN*
$$\frac{(Q)\ \Gamma, x : \sigma \vdash a : \sigma' \qquad Q' \sqsubseteq Q}{(Q)\ \Gamma \vdash \texttt{fun}\ (x : \exists\,(Q')\ \sigma)\ a : \forall\,(\alpha = \sigma)\ \forall\,(\alpha' \geq \sigma')\ \alpha \to \alpha'}$$

This rule is actually simpler than the derived annotation rule suggested by lemma 5, because instantiation is left to each occurrence of the annotated program variable $x$ in $a$.

The derived reduction is $(\texttt{fun}\ (x : \exists\,(Q)\ \sigma)\ a)\ v \xrightarrow{\beta'}$ `let` $x = (v : \exists\,(Q)\ \sigma)$ `in` $a$ and the expressions of the form `fun` $(x : \exists\,(Q)\ \sigma)\ a$ are values, indeed.

## 5.2 Expressiveness of annotations

We have seen that all ML programs can be written in ML$^{F}$ without any annotation at all. Annotations allows for typing in ML$^{F}$ all programs that are typable in System F, modulo a straightforward translation of types and terms. The encoding is given in Appendix E.

## 6 Discussion

### 6.1 Subsystems of ML$^{F}$ (*Conjectures*)

**Note:** *Claims of this section have only been sketched informally—their formal verification is planned as future work. The conjectures made at the end of the section are really open questions.*

Types whose flexible bounds are always $\bot$ are called F-types (they are the translation of types of System F, as defined in Appendix E). Types with restricted flexible bounds, *i.e.* of the form $\forall\,(\alpha \geq \sigma)\ \tau$ where $\sigma$ is not equivalent to a monotype nor to $\bot$, have been introduced to factor out choices during type inference. Such types are indeed used in a derivation of `let` $f$ `=` `choose id in` $(f$ `auto`$)$ $(f$ `succ`$)$. However, should these also be allowed as annotations? A term of ML$^{F}$ is shallow if it only contains F-type annotations. A type of ML$^{F}$ is *shallow* if its rigid bounds are F-types. More generally, a prefix (resp. typing context, judgment, or derivation) is shallow whenever it contains only shallow types and terms. Actually, any valid shallow judgment $(Q)\ \Gamma \vdash a : \tau$ has a shallow derivation. (The same property holds for instance and sharing relations judgments.) This suggests a restriction Shallow-ML$^{F}$ of ML$^{F}$ composed of shallow terms that admit shallow typing judgments. Moreover, subject reduction holds for Shallow-ML$^{F}$.

Expressiveness of ML$^{F}$ with and without let-bindings are equivalent, because all types can be used as annotations. However, this is no longer true for Shallow-ML$^{F}$, since shallow-types that are not F-types cannot be used as annotations. Therefore, we also consider the restriction of Shallow-ML$^{F}$ to programs without let-bindings, called Shallow-F.

The encoding of System-F into ML$^{F}$ is actually an encoding into Shallow-F. Conversely, all programs typable into Shallow-F are also typable in System-F. Hence Shallow-F and System-F have the same expressiveness. Terms of Shallow-F still require fewer type annotations. As a consequence terms of ML can of course be typed in Shallow-F. However, this may require annotations. Fortunately, all terms of ML can be typed in Shallow-ML$^{F}$ without any annotation at all.

We conjecture that there exists a term of Shallow-ML$^{F}$ that cannot be typed in Shallow-F even after removing or inserting any number of F-type annotations. Still, Shallow-ML$^{F}$ remains a second-order system and in that sense should not be *significantly* more expressive that System F. In particular, we conjecture that the term $(\texttt{fun}\ (y)\ z\ (y\ I)\ (y\ K))\ (\texttt{fun}\ (x)\ x\ x)$ that is typable in F$^{\omega}$ but not in F [GR88] is not typable in Shallow-ML$^{F}$ (nor in ML$^{F}$).

As in ML, reducing all let-bindings in a term of Shallow-ML$^{F}$ produces a term of Shallow-F. Hence, terms of Shallow-ML$^{F}$ are strongly normalizable. We conjecture that this is also the case for ML$^{F}$.

### 6.2 Simple language extensions

Because the language is parameterized by constants, which can be used either as constructors or primitive operations, the language can import foreign functions defined via appropriate $\delta$-rules. These could include primitive types (such as integers, strings, *etc.*) and operations over them. More generally, sums and products, as well as predefined data-types can also be treated in this manner, but some (easy) extension is required to declare new data-types within the language itself.

The value restriction of polymorphism [Wri95] that allows for safe mutable data-structures in ML should carry over to ML$^{\mathsf{F}}$ by allowing only rigid bounds that appear in the type of expansive expressions to be generalized. However, this solution is likely to be disappointing in ML$^{\mathsf{F}}$, as it is in Poly-ML, which uses polymorphism extensively. A relaxation of the value-only restriction has been recentlty proposed [Gar02]. It gave satisfactory results in the context of Poly-ML and we can expect similar benefits for ML$^{\mathsf{F}}$.

## 6.3 Related works

Our work is related to all other works that aim at some form of type inference in the presence of higher-order types, at least in their goals. The closest of them is unquestionably Poly-ML [GR99], with which close connections have already been made. Poly-ML also subsumes previous proposals that encapsulate first-class polymorphic values within datatypes [Rém94]. The proposal [OL96] also falls into this category; however, a side mechanism simultaneously allows a form of toplevel rank-2 quantification, which is not covered by Poly-ML but is, we think, subsumed by ML$^{\mathsf{F}}$.

Rank-2 polymorphism actually allows for full type inference [KW94, Jim95]. However, the algorithm is defined by reduction on source terms and is not very intuitive. Rank-2 polymorphism has also been incorporated in the Hugs implementation of Haskell [Mar02], but with explicit type annotations. The GHC implementation of Haskell has recently been released with second-order polymorphism at arbitrary ranks [GHC02]; however, types at rank 2 or higher must be given explicitly and the interaction of annotations with implicit types remains unclear. Furthermore, to the best of our knowledge, this has not yet been formalized. Indeed, type inference is undecidable as soon as universal quantifiers may appear at rank 3 [KW99].

Although our proposal relies on the ML let-binding mechanism to introduce implicit polymorphism—even for annotations—and flexible bounds in types to factorize all ways of obtaining type instances, rather than on intersection types, there may still be some connection with intersection types [Jim96], which we would like to explore. Our treatment of annotations as “unsharing” primitives also resembles retyping functions (functions whose type-erasure $\eta$-reduces to the identity) [Mit88]. However, our annotations are explicit and contain only certain forms of retyping functions. Type inference for System F modulo $\eta$-expansion is known to be undecidable as well [Wel96].

Several people have considered partial type inference for System F [JWOG89, Boe85, Pfe93] and stated undecidability results for some particular variants that in all cases amount—directly or indirectly—to permit (and so force) inference of the type of at least one variable that can be used in a polymorphic manner, which we avoid.

Second-order unification, although known to be undecidable, has been used to explore the practical effectiveness of type inference for System F [Pfe88]. Despite our opposite choice, that is not to support second-order unification, there are at least two comparisons to be made. Firstly, their proposal does not cover the language ML *per se*, but only the $\lambda$-calculus, since let-bindings are expanded prior to type inference. Indeed, ML is not the simply-typed $\lambda$-calculus and type inference in ML cannot, *in practice*, be reduced to type inference in the simply-typed $\lambda$-calculus after expansion of let-bindings. Secondly, one proposal seems to require annotations exactly where the other can skip them: in [Pfe88], markers (but no type) annotations must replace type-abstraction and type-application nodes; conversely, this information is omitted in ML$^{\mathsf{F}}$, but instead, explicit type information must remain for (some) arguments of $\lambda$-abstractions.

Our proposal is implicitly parameterized by the type instance relation and its corresponding unification algorithm. Thus, most of the technical details can be encapsulated within the instance relation and its properties. We would like to understand our unification algorithm as a particular case of second-order unification. One step in this direction would be to consider a modular constraint-based presentation of second-order unification such as [DHKP96]. Flexible bounds might capture, within principal types, what constraint-based algorithms capture as partially unresolved multi-sets of unification constraints. Another example of restricted unification within second-order terms is unification under a mixed prefix [Mil92]. However, the notion of prefix in their work is rather different and does not express “sharing”.

Actually, none of the above works did consider subtyping at all. This is a significant difference with proposals based on local type inference [Car93, PT98, OZZ01] where subtyping is a prerequisite. The addition of subtyping to our framework remains to be explored.

Furthermore, beyond its treatment of subtyping, local type inference also brings the idea that explicit type annotations can be propagated up and down the source tree according to fixed well-defined rules, which, at least intuitively, could be understood as a preprocessing of the source term. Such a mechanism is being used in the GHC Haskell compiler, and could in principle be added on top of ML$^{\mathsf{F}}$ as well.

## Conclusions

We have proposed an integration of ML and System F that combines the convenience of type inference as present in ML and the expressiveness of second-order polymorphism. Type information is only required for arguments of functions that are used polymorphically in their bodies. This specification should be intuitive to the user. Besides, it is modular, since annotations depend more on the behavior of the code than on the context in which the code is placed; in particular, functions that only carry polymorphism without using it can be left unannotated.

The obvious potential application of our work is to extend ML-like languages with second-order polymorphism while keeping full type inference for a large subset of the language, containing at least all ML programs. However, further investigations are needed beforehand, in particular regarding the syntactic-value polymorphism restriction.

Furthermore, on the theoretical side, we wish to better understand the concept of "first-order unification of second-order terms", and, if possible, to confine it to an instance of second-order unification. We would also like to give logical meaning to our types and to the sharing and instance relations.

## References

[Boe85] H.-J. Boehm. Partial polymorphic type inference is undecidable. In *26th Annual Symposium on Foundations of Computer Science*, pages 339–345, Los Angeles, Ca., USA, October 1985. IEEE Computer Society Press.

[Car93] Luca Cardelli. An implementation of FSub. Research Report 97, Digital Equipment Corporation Systems Research Center, 1993.

[Cos95] Roberto Di Cosmo. *Isomorphisms of Types: from lambda-calculus to information retrieval and language design*. Birkhauser, 1995.

[DHKP96] Gilles Dowek, Thérèse Hardin, Claude Kirchner, and Frank Pfenning. Higher-order unification via explicit substitutions: the case of higher-order patterns. In M. Maher, editor, *Joint international conference and symposium on logic programming*, pages 259–273, 1996.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the Ninth ACM Conference on Principles of Programming Langages*, pages 207–212, 1982.

[Gar02] Jacques Garrigue. Relaxing the value-restriction. Presented at the third Asian workshop on Programmaming Llanguages and Systems (APLAS), 2002.

[GHC02] The GHC Team. *The Glasgow Haskell Compiler User's Guide, Version 5.04*, 2002. Chapter *Arbitrary-rank polymorphism*.

[GR88] P. Giannini and S. Ronchi Della Rocca. Characterization of typings in polymorphic type discipline. In *Third annual Symposium on Logic in Computer Science*, pages 61–70. IEEE, 1988.

[GR99] Jacques Garrigue and Didier Rémy. Extending ML with semi-explicit higher-order polymorphism. *Journal of Functional Programming*, 155(1/2):134–169, 1999. A preliminary version appeared in TACS'97.

[Jim95] Trevor Jim. Rank-2 type systems and recursive definitions. Technical Report MIT/LCS/TM-531, Massachusetts Institute of Technology, Laboratory for Computer Science, November 1995.

[Jim96] Trevor Jim. What are principal typings and what are they good for? In ACM, editor, *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 42–53, 1996.

[JWOG89] Jr. James William O'Toole and David K. Gifford. Type reconstruction with first-class polymorphic values. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, Oregon, June 1989. ACM. also in ACM SIGPLAN Notices 24(7), July 1989.

[KW94] Assaf J. Kfoury and Joe B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order lambda -calculus. In *ACM Conference on LISP and Functional Programming*, pages 196–207, 1994.

[KW99] Assaf J. Kfoury and Joe B. Wells. Principality and decidable type inference for finite-rank intersection types. In *ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas*, pages 161–174, New York, NY, January 1999. ACM.

[LDG$^+$02] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system, documentation and user's manual - release 3.05. Technical report, INRIA, July 2002. Documentation distributed with the Objective Caml system.

[LO94] Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, September 1994.

[Mar02] Mark P Jones, Alastair Reid, the Yale Haskell Group, and the OGI School of Science & Engineering at OHSU. An overview of hugs extensions. Available electronically, 1994-2002.

[Mil92] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.

[Mit88] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 2/3(76):211–249, 1988.

[OL96] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Proceedings of the 23rd ACM Conference on Principles of Programming Languages*, pages 54–67, January 1996.

[OZZ01] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. *ACM SIGPLAN Notices*, 36(3):41–53, March 2001.

[Pfe88] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 153–163, Snowbird, Utah, July 1988. ACM Press.

[Pfe93] Frank Pfenning. On the undecidability of partial polymorphic type reconstruction. *Fundamenta Informaticae*, 19(1,2):185–199, 1993. Preliminary version available as Technical Report CMU-CS-92-105, School of Computer Science, Carnegie Mellon University, January 1992.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *Proceedings of the 25th ACM Conference on Principles of Programming Languages*, 1998. Full version in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1), January 2000, pp. 1–44.

[Rém94] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 321–346. Springer-Verlag, April 1994.

[Wel94] J. B. Wells. Typability and type checking in the second order $\lambda$-calculus are equivalent and undecidable. In *Ninth annual IEEE Symposium on Logic in Computer Science*, pages 176–185, Paris, France, July 1994.

[Wel96] Joe B. Wells. *Type Inference for System F with and without the Eta Rule*. PhD thesis, Boston University, 1996.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

## A Occurrences

**Occurrences and free variables** An occurrence is a sequence of natural numbers. We write $k^n$ for the sequence $k, \ldots, k$ of length $n$ (and, in particular, $k^0$ is $\epsilon$). The *projection* is a function mapping pairs $\sigma/u$ composed of a type $\sigma$ and an occurrence $u$ to $\{\bot\} \cup \vartheta \cup \mathcal{G}$ and defined inductively as follows:

$$\bot/\epsilon = \bot \qquad \alpha/\epsilon = \alpha \qquad (g^n\ \tau_1\ ..\ \tau_n)/\epsilon = g$$

$$\frac{1 \leqslant i \leqslant n}{(g^n\ \tau_1\ ..\ \tau_n)/iu = \tau_i/u} \qquad \frac{\sigma_1/u_1 \neq \alpha}{(\forall\,(\alpha \diamond \sigma_2)\ \sigma_1)/u_1 = \sigma_1/u_1}$$

$$\frac{\sigma_1/u_1 = \alpha}{(\forall\,(\alpha \diamond \sigma_2)\ \sigma_1)/u_1u_2 = \sigma_2/u_2}$$

We write $\sigma/$ the function $u \mapsto \sigma/u$. We call *occurrences of a type scheme* $\sigma$ the domain of the function $\sigma/$, which we abbreviate as $\mathsf{dom}(\sigma)$.

**Free type variables** A type variable $\alpha$ is *free* in $\sigma$ if there exists an occurrence $u$ such that $\sigma/u$ is $\alpha$. We write $\mathsf{ftv}(\sigma)$ the set of *free type variables* of $\sigma$. A type scheme is *closed* if it has no free type variable. When $\forall\,(Q)\ \sigma$ is closed, we also say that $\sigma$ is closed under $Q$.

Occurrences—and therefore free type variables—are stable under equivalence, as stated by the following lemma.

**Lemma 9** *If $(Q)\ \sigma \equiv \sigma'$ holds, then $(\forall\,(Q)\ \sigma)/$ and $(\forall\,(Q)\ \sigma')/$ are equal.*

Given a set $\bar{\alpha}$ of variables, we define the type $\nabla_{\bar{\alpha}}$ as $\alpha_1 \to \ldots \to \alpha_n \to \texttt{unit}$. By extension, we note $\nabla_Q$ for $\nabla_{\mathsf{dom}(Q)}$.

**Domain of prefixes** As seen earlier, if $Q$ is $(\alpha_1 \diamond_1 \sigma_1, \ldots \alpha_n \diamond_n \sigma_n)$, then its domain, written $\mathsf{dom}(Q)$, is the set $\{\alpha_1, \ldots \alpha_n\}$. In some proofs, we need to capture the notion of useful domain of a prefix $Q$. If $I$ is a set of type variables, we define $\mathsf{dom}(Q/I)$ as follows:

$$\mathsf{dom}(\emptyset/I) \triangleq \emptyset \qquad \mathsf{dom}((\alpha \diamond \sigma, Q)/I) \triangleq$$

$$\begin{cases} \{\alpha\} \cup \mathsf{dom}(Q/I) & \text{when } \alpha \in \mathsf{ftv}(\forall\,(Q)\ \nabla_I) \\ \mathsf{dom}(Q/I) & \text{otherwise} \end{cases}$$

We also write $\mathsf{dom}(Q/\sigma)$ to mean $\mathsf{dom}(Q/\mathsf{ftv}(\sigma))$. Intuitively, this corresponds to the domain of $Q$ which is useful for $\sigma$. As an example, if $Q'$ corresponds to $Q$, where all bindings not in $\mathsf{dom}(Q/\sigma)$ have been removed, then we have $\forall\,(Q)\ \sigma \equiv \forall\,(Q')\ \sigma$.

## B Unification algorithm

Interestingly, the unification algorithm follows the structure of first-order unification, despite the presence of polytypes. Indeed, the computation of the unifying substitution is replaced by the computation of a unifying prefix. While in the first-order case, bounds of the prefix could only be $\perp$ or monotypes, they can, in the general case, also be polytypes. One consequence is that the algorithm must consider a few more cases. Another consequence is that a polytype bound of a variable may have to be instantiated to a monotype. This requires to extend the prefix with more bindings. For instance, instantiating the bound $\forall\,(Q')\ \tau$ in the prefix $(Q, \alpha \geq \forall\,(Q')\ \tau)$ leads to the prefix $(QQ', \alpha \geq \tau)$—applying a rule similar to Rule I-UP but for prefixes. A *rigid* prefix $Q$ is a prefix with only rigid bounds. Two auxiliary operators are used to ease such manipulation of prefixes. The binary operator $Q\!\uparrow\!\bar{\alpha}$ performs only commutations of binders in $Q$ and returns a pair of prefixes $(Q_1, Q_2)$ such that $Q_1Q_2$ is equivalent to $Q$ and differ only by reordering of bindings, $\bar{\alpha} \subseteq \mathsf{dom}(Q_1)$, and $\mathsf{dom}(Q_1/\bar{\alpha}) = \mathsf{dom}(Q_1)$.

**Definition 2** The *update* of a prefix $Q$ by a binding $(\alpha \diamond \sigma)$ such that $\alpha \in \mathsf{dom}(Q)$, written $Q \Leftarrow (\alpha \diamond \sigma)$, is a prefix $(Q_0, \alpha \diamond \sigma, Q_1)$ such that $(Q_0, \alpha \diamond_1 \sigma_1, Q_1)$ is a reordering of toplevel bindings of $Q$ that preserves equivalence, $\mathsf{dom}(Q_1) \cap \mathsf{ftv}(\sigma) = \emptyset$ and $Q_1$ is as small as possible. It is undefined if $\mathsf{ftv}(\sigma) \cap \mathsf{dom}(Q_1) \neq \emptyset$ in all such decompositions. □

The unification algorithm *unify* $(Q, \tau, \tau')$ takes as input a prefix $Q$ and two types and either returns a prefix that unifies these types (as described by lemmas 2 and 3) or fails. The algorithm is described in Figure 8. For the sake of comparison with ML, lines marked with • could be removed in the absence of polytypes. Two steps at the end of subcase $(\alpha_1, \alpha_2)$ use an auxiliary algorithm $(Q)\ \sigma \not\sqsubseteq \sigma'$ to check, given $Q, \sigma$ and $\sigma'$ such that $(Q)\ \sigma \sqsubseteq \sigma'$ holds, whether $(Q)\ \sigma \sqsubseteq \sigma'$ holds.

**Lemma 10 (Completeness of unification)** *Assume* $\mathsf{ftv}(\tau, \tau') \subseteq I$. *If there exists $Q_2$ such that $Q_1 \sqsubseteq^I Q_2$ and $(Q_2)\ \tau \equiv \tau'$ hold, then unify $(Q_1, \tau, \tau')$ succeeds with $Q_1'$ and we have $Q_1' \sqsubseteq^I Q_2$.*

## C Standard properties

The following lemma states two standard properties of typing judgments.

**Lemma 11** *The following rules are admissible:*

STRENGTHEN
$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \Gamma' \sqsubseteq \Gamma}{(Q)\ \Gamma' \vdash a : \sigma}$$

WEAKEN
$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad Q \sqsubseteq Q'}{(Q')\ \Gamma \vdash a : \sigma}$$

In a judgment $(Q)\ \Gamma \vdash a : \sigma$, the type scheme $\sigma$ can be weakened as described by Rule INST. Conversely, the context $\Gamma$ can be strengthened, as described by Rule STRENGTHEN. This rule uses the instance relation between typing contexts $(Q)\ \Gamma \sqsubseteq \Gamma'$, which is an abbreviation for:

$$\forall z \in \mathsf{dom}(\Gamma), z \in \mathsf{dom}(\Gamma') \wedge (Q)\ \Gamma(z) \sqsubseteq \Gamma'(z)$$

In addition to weakening the type or strengthening the context, the whole judgment can be instantiated. In ML, this is expressed by stability of typing judgments by substitutions. Here, this is modeled by instantiating the prefix of the typing judgment, as described by rule WEAKEN, which is consistent with viewing prefixes as a generalization of substitutions.

The substitutivity lemma is key to subject reduction.

**Lemma 12 (Substitutivity)** *If $(Q)\ \Gamma, x : \sigma \vdash a_0 : \sigma_0$ and if $(Q)\ \Gamma \vdash a : \sigma$, then $(Q)\ \Gamma \vdash a_0[a/x] : \sigma_0$.*

Finally, we state the equivalence between the syntax-directed and original presentations of the typing rules.

**Lemma 13** *The judgment $(Q)\ \Gamma \vdash a : \sigma$ hold if and only if there exists $\sigma'$ such that both judgments $(Q)\ \Gamma \overset{\triangle}{\vdash} a : \sigma'$ and $(Q)\ \sigma' \sqsubseteq \sigma$ hold.*

## D Type inference algorithm

## E Encoding System F into ML$^\mathrm{F}$

The types, terms, and typing contexts of system $F$ are given below:

$$\begin{aligned} t &::= \alpha \mid t \to t \mid \forall\, \alpha \cdot t \\ M &::= x \mid M\ M \mid \mathtt{fun}\ (x : t)\ M \mid \mathtt{Fun}\ (\alpha)\ M \mid M\ t \\ A &::= \emptyset \mid A, x : t \mid A, \alpha \end{aligned}$$

The translation of types of System F into ML$^\mathrm{F}$ types uses auxiliary rigid bindings for arrow types. This ensures that there are no inner polytypes left in the result of the translation—which would otherwise be ill-formed. Quantifiers that are present in the original type are translated to unconstrained bounds.

$$[\![\alpha]\!] = \alpha \qquad\qquad [\![\forall\, \alpha \cdot t]\!] = \forall\,(\alpha)\ [\![t]\!]$$

$$[\![t_1 \to t_2]\!] = \forall\,(\alpha_1 = [\![t_1]\!])\ \forall\,(\alpha_2 = [\![t_2]\!])\ \alpha_1 \to \alpha_2$$

In order to state the correspondence between typing judgments, we must also translate typing environments. We write $A \vdash M : t$ to mean that $M$ has type $t$ in environment $A$ in System F. The translation of $A$, written $[\![A]\!]$, returns a pair $(Q)\ \Gamma$ of a prefix and a typing environment and is defined inductively as follows:

$$[\![\emptyset]\!] = ()\ \emptyset \qquad\qquad \frac{[\![A]\!] = (Q)\ \Gamma}{[\![A, x : t]\!] = (Q)\ \Gamma, x : [\![t]\!]}$$

$$\frac{[\![A]\!] = (Q)\ \Gamma \qquad \alpha \notin \mathsf{dom}(Q)}{[\![A, \alpha]\!] = (Q, \alpha)\ \Gamma}$$

The translation of System F terms into ML$^\mathrm{F}$ terms forgets type abstraction and type applications, and translates types in term-abstractions.

$$[\![\mathtt{Fun}\ (\alpha)\ M]\!] = [\![M]\!] \qquad [\![M\ t]\!] = [\![M]\!] \qquad [\![x]\!] = x$$

$$[\![M\ M']\!] = [\![M]\!]\ [\![M']\!]$$

$$[\![\mathtt{fun}\ (x : t)\ M]\!] = \mathtt{fun}\ (x : [\![t]\!])\ [\![M]\!]$$

Finally, we can state the following lemma:

**Lemma 14** *For any closed typing environment $A$ (that does not bind the same type variable twice), term $M$ and type $t$ of system $F$ such that $A \vdash M : t$, there exists a derivation $(Q)\ \Gamma \vdash [\![M]\!] : \tau$ such that $(Q)\ \Gamma = [\![A]\!]$ and $[\![t]\!] \sqsubseteq \tau$.*

Note that translated terms contain strictly fewer annotations than original terms—a property that was not true in Poly-ML. Moreover, some annotations provided by the translation are superfluous.

**Figure 8. Unification algorithm**

The recursive algorithm *unify* $(Q, \tau'', \tau''')$ first rewrites all bounds of $Q$ in normal form and proceeds by case analysis on $(\tau'', \tau''')$:

**Case** $(\alpha, \alpha)$: **return** $Q$.

**Case** $(g\ \tau_1^1\ ..\ \tau_1^n, g\ \tau_2^1\ ..\ \tau_2^n)$:

- ○ let $Q^1$ be $Q$;
- ○ let $Q^{i+1}$ be *unify* $(Q^i, \tau_1^i, \tau_2^i)$ for $1 \leqslant i \leqslant n$;
- ○ **return** $Q^{n+1}$.

**Case** $(g_1\ \tau_1^1\ ..\ \tau_1^p, g_2\ \tau_2^1\ ..\ \tau_2^q)$ with $g_1 \neq g_2$: **fail**.

**Case** $(\alpha, \tau)$ or $(\tau, \alpha)$ when $(\alpha \diamond \tau') \in Q$:
**return** *unify* $(Q, \tau, \tau')$.

**Case** $(\alpha, \tau)$ or $(\tau, \alpha)$ when $(\alpha \diamond \sigma) \in Q$
and $\tau \notin \mathsf{dom}(Q)$ and $\sigma \notin \mathcal{T}$

- • if $(\alpha = \bot) \in Q$, then **fail**.
- ○ if $\sigma$ is $\bot$ then **return** $Q \Leftarrow (\alpha = \tau)$.
- ○ let $\forall\,(Q')\ \tau'$ be $\sigma$ with $\mathsf{dom}(Q)$, $\mathsf{dom}(Q')$ disjoint;
- • if $\diamond$ is $=$, check that $Q'$ is rigid, otherwise **fail**.
- ○ let $Q''$ be $(QQ') \Leftarrow (\alpha = \tau')$;
- ○ **return** *unify* $(Q'', \alpha, \tau)$.

**Case** $(\alpha_1, \alpha_2)$ when $\alpha_1 \neq \alpha_2$ and $(\alpha_1 \diamond_1 \sigma_1) \in Q$ and $(\alpha_2 \diamond_2 \sigma_2) \in Q$
and $\sigma_1$, $\sigma_2$ are not in $\mathcal{T}$.

- ○ let $\diamond$ be either $>$, if both $\diamond_1$ and $\diamond_2$ are $>$, and $=$ otherwise;
- ○ if $\sigma_1$ and $\sigma_2$ are $\bot$ then **return** $Q \Leftarrow (\alpha_1 \diamond \bot) \Leftarrow (\alpha_2 = \alpha_1)$;
- • if $\sigma_i$ is $\bot$ and $\diamond_i$ is $=$ then **fail**;
- • if $\sigma_1$ is $\bot$ then **return** $Q \Leftarrow (\alpha_1 = \alpha_2)$;
- • if $\sigma_2$ is $\bot$ then **return** $Q \Leftarrow (\alpha_2 = \alpha_1)$;
- • let $\forall\,(Q_1)\ \tau_1$ be $\sigma_1$ and $\forall\,(Q_2)\ \tau_2$ be $\sigma_2$ with $\mathsf{dom}(Q)$, $\mathsf{dom}(Q_1)$, $\mathsf{dom}(Q_2)$ disjoint;
- • let $Q_0$ be *unify* $(QQ_1Q_2, \tau_1, \tau_2)$;
- • let $(Q_3, Q')$ be $Q_0 {\uparrow} \mathsf{dom}(Q)$;
- • if $\diamond_1$ is $=$ and $(Q_3)\ \sigma_1 \not\equiv \forall\,(Q')\ \tau_1$ then **fail**;
- • if $\diamond_2$ is $=$ and $(Q_3)\ \sigma_2 \not\equiv \forall\,(Q')\ \tau_2$, then **fail**;
- • let $\sigma_3$ be $\forall\,(Q')\ \tau_1$;
- • **return** $(Q_3) \Leftarrow (\alpha_1 \diamond \sigma_3) \Leftarrow (\alpha_2 = \alpha_1)$.

**Figure 9. Algorithm** $\mathrm{W}^{\mathrm{F}}$

The algorithm *infer* $(Q, \Gamma, a)$ is defined by cases on expression $a$:

**Case** $x$ : **return** $Q, \Gamma(x)$

**Case** `fun` $(x)\ a$ :

```
– let $Q_1 = (Q, \alpha > \bot)$ with $\alpha \notin \mathsf{dom}(Q)$ in
– let $(Q_2, \sigma) = \mathit{infer}\ (Q_1, \Gamma, x : \alpha, a)$ in
– let $\beta \notin \mathsf{dom}(Q_2)$ and $(Q_3, Q_4) = Q_2 \uparrow \mathsf{dom}(Q)$ in
– return $Q_3, \forall\,(Q_4)\ \forall\,(\beta \geq \sigma)\ \alpha \to \beta$
```

**Case** $a\ b$ :

```
– let $(Q_1, \sigma_a) = \mathit{infer}\ (Q, \Gamma, a)$ in
– let $(Q_2, \sigma_b) = \mathit{infer}\ (Q_1, \Gamma, b)$ in
– let $\alpha_a, \alpha_b, \beta \notin \mathsf{dom}(Q_2)$ in
– let $Q_3 = \mathit{unify}\ ((Q_2, \alpha_a \geq \sigma_a, \alpha_b \geq \sigma_b, \beta \geq \bot),$
                                   $\alpha_a, \alpha_b \to \beta)$ in
– let $(Q_4, Q_5) = Q_3 \uparrow \mathsf{dom}(Q)$ in
– return $(Q_4, \forall\,(Q_5)\ \beta)$
```

**Case let** $x = a_1$ **in** $a_2$ :

```
– let $(Q_1, \sigma_1) = \mathit{infer}\ (Q, \Gamma, a_1)$ in
– return infer $(Q_1, (\Gamma, x : \sigma_1), a_2)$
```