ML^{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. Expressions may contain second-order type annotations. Every typable expression admits a principal type, which however depends on type annotations. Principal types capture all other types that can be obtained by implicit type instantiation and they can be inferred. All expressions of ML are well-typed without any annotations. All expressions of System F can be mechanically encoded into ML^{F} by dropping all type abstractions and type applications, and injecting types of lambda-abstractions into ML^{F} types. Moreover, only parameters of lambda-abstractions that are used polymorphically need to remain annotated.

Categories and Subject Descriptors: D.3.3 Language Constructs and Features.

General Terms: Theory, Languages.

Keywords: Type Inference, First-Class Polymorphism, Second-Order Polymorphism, System F, ML, Type Annotations.

The quest for type inference with first-class polymorphic types

Programming languages considerably benefit from static type-checking. In practice however, types may sometimes trammel programmers, for two opposite reasons. 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 relieves the programmer from writing such details and simultaneously lightens programs. In parallel, more expressive type systems have been developed, so that programmers are less often exposed to their limitations.

Unfortunately, those two situations are often conflicting. Expressive type systems tend to require an unbearable amount of type decorations, thus many of them only remained at the status of prototypes. Indeed, full type inference for System F is undecidable [26]. 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 [4] 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 [25, 20, 7] and, even though these may not occur too frequently, ML does not offer any reasonable alternative. (The inconvenience is often underestimated, since the lack of a full-fledged language to experiment with first-class polymorphism insidiously keeps programmers thinking in terms of ML polymorphism.)

A first approach is to extend ML with first-class second-order polymorphism [15, 25, 20, 7]. 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 [2], 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* [24], recently improved to *colored* local type inference [21]. 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

ICFP'03, August 25–29, 2003, Uppsala, Sweden.

already quite happy with ML[1]. We build on some previous work [7], which has been used to add polymorphic methods to OCaml [16]. Here, we retain the same primary goal, that is to type all expressions of System-F, providing explicit annotations when needed, and to keep all expressions of ML unannotated. In addition, we aim at the elimination of all backward coercions from polymorphic types to ML-types. In particular, our goal is not to guess polymorphic types.

Our track

Church's style System-F and ML are quite different in nature. In ML, the elimination of polymorphism is implicitly performed at every use occurrence of a variable bound with a polymorphic type $\forall\bar{\alpha}.\tau$, which can then be given any instance, of the form $\tau[\bar{\tau}/\bar{\alpha}]$. Indeed, a polymorphic type somehow represents the set of its instances. This induces an instance relation between polymorphic types themselves. For example, the (polymorphic) type $\forall(\alpha)\ \alpha \to \alpha$ is said to be more general than $\forall(\alpha)\ \forall(\beta)\ (\alpha \to \beta) \to (\alpha \to \beta)$ and we write $\forall(\alpha)\ \alpha \to \alpha \preccurlyeq \forall(\alpha)\ \forall(\beta)\ (\alpha \to \beta) \to (\alpha \to \beta)$ because all instances of the latter are also instances of the former.

Conversely, in Church's style System F, a type $\forall(\alpha)\ \alpha \to \alpha$ only stands for itself (modulo renaming of bound variables) and the elimination of polymorphism must be performed explicitly by type application (and abstraction) at the source level. A counter-part in System F is that bound type variables may be instantiated by polymorphic types, allowing for expressive impredicative second-order types. For example, an expression of type $\forall(\alpha)\ \alpha \to \alpha$ can be given type $(\forall(\beta)\ \beta \to \beta) \to (\forall(\beta)\ \beta \to \beta)$ by an explicit type-application to $\forall(\beta)\ \beta \to \beta$.

Unfortunately, combining implicit instantiation of polymorphic types with second-order types raises conflicts almost immediately. For illustration, consider the application of the function `choose`, defined as $\lambda(x)\ \lambda(y)$ `if` *true* `then` x `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 $\mathtt{id}_\alpha$ for $\alpha \to \alpha$ and $\sigma_{\mathtt{id}}$ for $\forall(\alpha)\ \mathtt{id}_\alpha$. Should `choose id` have type σ_1 equal to $\forall(\alpha)\ \mathtt{id}_\alpha \to \forall(\alpha)\ \mathtt{id}_\alpha$, obtained by keeping the type of `id` uninstantiated? Or, should it have type σ_2 equal to $\forall(\alpha)\ (\mathtt{id}_\alpha \to \mathtt{id}_\alpha)$, obtained by instantiating the type of `id` to the monomorphic type $\mathtt{id}_\alpha$ and generalizing α only at the end? Indeed, both σ_1 and σ_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 $\lambda(x : \sigma_{\mathtt{id}})\ x\,x$ can be typed with σ_1, as `choose id`, but not with σ_2; otherwise `auto` could be applied, for instance, to the successor function, which would lead to a runtime error. Hence, σ_1 cannot be safely coerced to σ_2. Conversely, however, there is a retyping function—a function whose type erasure η-reduces to the identity [19]—from type σ_2 to type σ_1, namely, $\lambda(g : \sigma_2)\ \lambda(x : \sigma_{\mathtt{id}})\ \lambda(\alpha)\ g\ \alpha\ (x\ \alpha)$. Actually, σ_2 is a principal type for `choose id` in $F^{\eta *}$ (System F closed by η-expansion) [19].

While the argument of `auto` must be at least as polymorphic as $\sigma_{\mathtt{id}}$, the argument of the function `choose id` need not be polymorphic: it may be any instance τ of $\sigma_{\mathtt{id}}$ and the type of the return value is then τ. We could summarize these constraints by saying that:

$$\begin{aligned} \mathtt{auto} :&\quad \forall(\alpha = \sigma_{\mathtt{id}})\ \alpha \to \alpha \\ \mathtt{choose\ id} :&\quad \forall(\alpha \geq \sigma_{\mathtt{id}})\ \alpha \to \alpha \end{aligned}$$

The type given to `choose id` captures the intuition that this application has type $\tau \to \tau$ for any instance τ of $\sigma_{\mathtt{id}}$. This form of quantification allows to postpone the decision of whether $\sigma_{\mathtt{id}}$ should be instantiated as soon as possible or kept polymorphic as long as possible. The bound of α in $\forall(\alpha \geq \sigma_{\mathtt{id}})\ \alpha \to \alpha$, which is said to be *flexible*, can be weakened either by instantiating $\sigma_{\mathtt{id}}$ or by replacing $\geq$ by $=$. Both forms of weakening can be captured by an appropriate instance relation $\preccurlyeq$ between types. In a binder of the form $(\alpha = \sigma)$ the bound σ, which is said to be *rigid*, cannot be instantiated any longer. Intuitively, the type $\forall(\alpha = \sigma)\ \sigma'$ stands for the System-F type $\sigma'[\sigma/\alpha]$.

Finally, both `choose id succ` and `choose id auto` are well-typed, taking `int` → `int` or $\sigma_{\mathtt{id}}$ for the type of α, respectively. In fact, the type $\forall(\alpha \geq \sigma_{\mathtt{id}})\ \alpha \to \alpha$ happens to be a principal type for `choose id` in ML^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 as "α is any instance of $\sigma_{\mathtt{id}}$", are kept unresolved. In short, ML^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.

A technical road-map

The instantiations between types used above remain to be captured formally within an appropriate relation $\preccurlyeq$. Indeed, the instance relation plays a crucial role in type inference, via a typing rule INST stating that any expression a of type σ in a context Γ has also type σ' in the same context whenever $\sigma \preccurlyeq \sigma'$. Intuitively, the larger the relation $\preccurlyeq$ is, the more flexibility is left for inference, and, usually, the harder the inference algorithm is.

Unsurprisingly, the "smallest reasonable relation" $\preccurlyeq$ that validates all the instantiations used above leads to undecidable type inference, for full type inference in System-F is undecidable. Still, the relation $\preccurlyeq$ induces an interesting variant UML^F that has the same expressiveness as ML^F but requires no type annotations at all. Fortunately, the relation $\preccurlyeq$ can be split into a composition of relations $\exists\sqsubseteq\exists$ where uses of the relation $\sqsubseteq$ can be inferred as long as all applications of $\exists$ are fully explicit: this sets a clear distinction between explicit and inferred type information, which is the essence of ML^F.

Unfortunately, subject reduction does not hold in ML^F for a simple notion of reduction (non local computation of annotations would be required during reduction). Thus, we introduce an intermediate variant ML$^F_\star$ where only place holders for $\exists$ are indicated. For example, using the symbol $\star$ in place of polytypes, $\lambda(x : \star)\ x\,x$ belongs to ML$^F_\star$ since $\lambda(x : \sigma_{\mathtt{id}})\ x\,x$ belongs to ML^F (and of course, $\lambda(x)\ x\,x$ belongs to UML^F). Subject reduction and progress are proved for ML$^F_\star$ and type soundness follows for ML$^F_\star$ and, indirectly, for ML^F.

In fact, we abstract the presentation of ML$^F_\star$ over a collection of primitives so that ML^F can then be embedded into ML$^F_\star$ by treating type-annotations as an appropriate choice of primitives and disallowing annotation place holders in source terms. Thus, although our practical interest is the system ML^F, most of the technical developments are pursued in ML$^F_\star$.

Unsurprisingly, neither UML^F nor ML$^F_\star$ admits principal types. Conversely, *every expression typable in ML^F admits a principal type*. Of course, principal types depend on type annotations in

---

[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

$$\tau ::= \alpha \mid g^n\ \tau_1\ ..\ \tau_n \qquad \text{Monotypes}$$
$$\sigma ::= \tau \mid \bot \mid \forall(\alpha \geq \sigma)\ \sigma \mid \forall(\alpha = \sigma)\ \sigma \qquad \text{Polytypes}$$

the source term. More precisely, if an expression is not typable in ML^{F}, it may sometimes be typable by adding extra type annotations. Moreover, two different type annotations may lead to two incomparable principal types. As an example, the expression $\lambda(x)\ x\ x$ is not typable in ML^{F}, while both expressions $\lambda(x : \forall\alpha.\alpha)\ x\ x$ and $\lambda(x : \forall\alpha.\alpha \to \alpha)\ x\ x$ are typable, with incomparable types. Adding a (polymorphic) type annotation to a typable expression may also lead to a new type that is not comparable with the previous one. This property should not be surprising since it is inherent to second-order polymorphism, which we keep explicit—remember that we only infer first-order polymorphism in the presence of second-order types. Still, the gain is the elusion of most type annotations, via the instance relation $\sqsubseteq$.

The paper is organized as follows. In Section 1, we describe types and instance relations $\sqsubseteq$ and $\sqsubseteq$. The syntax and the static and dynamic semantics of $\mathrm{ML}^{\mathrm{F}}_{\star}$ are described in Section 2. Section 3 presents formal properties, including type soundness for $\mathrm{ML}^{\mathrm{F}}_{\star}$ and type inference for ML^{F}. Section 4 introduces explicit type annotations. A comparison with System-F is drawn is Section 5. In Section 6, we discuss expressiveness, language extensions, and related works. For the sake of readability, unification and type inference algorithms have been moved to the appendices. Due to lack of space, all proofs are omitted.

"Monomorphic abstraction of polymorphic types"

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, $\lambda(x)\ x\ x$ cannot be typed in ML^{F}. In particular, we do not infer types for function arguments that are used polymorphically. Fortunately, such arguments can be annotated with a polymorphic 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, both `id auto` and `choose id auto` remain of type $\forall(\alpha = \sigma_{\mathtt{id}})\ \alpha \to \alpha$ (the type $\sigma_{\mathtt{id}}$ of `auto` is never instantiated) and neither `choose` nor `id` require any type annotation. Finally, polymorphic functions can be used by implicit instantiation, much as in ML.

To summarize, a key feature of ML^{F} is that type variables can *always* be implicitly instantiated by polymorphic types. This can be illustrated by the killer-app(lication) $(\lambda(x)\ x$ `id`$)$ `auto`. This expression is typable in ML^{F} as such, that is without any type application nor any type annotation—except, of course, in the definition of `auto` itself. In fact, a generalization of this example is the `app` function $\lambda(f)\ \lambda(x)\ f\ x$, whose ML^{F} principal type is $\forall(\alpha,\beta)\ (\alpha \to \beta) \to \alpha \to \beta$. It is remarkable that whenever $a_1\ a_2$ is typable in ML^{F}, so is `app` $a_1\ a_2$, without any type annotation nor any type application. This includes, of course, cases where a_1 expects a polymorphic value as argument, such as in `app auto id`. We find such examples quite important in practice, since they model iterators (*e.g.* `app`) applied to polymorphic functions (*e.g.* `auto`) over structures holding polymorphic values (*e.g.* `id`).

1 Types

1.1 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 $|g|$. To avoid degenerated cases, we assume that $\mathcal{G}$ contains at least a symbol of arity two (the infix arrow $\to$). We write g^n if g is of arity n. We also write $\bar{\tau}$ for tuples of types. The polytype $\bot$ corresponds to $(\forall\alpha.\alpha)$, intuitively. More precisely, it will be made equivalent to $\forall(\alpha \geq \bot)\ \alpha$.

We distinguish between *monotypes* and *polytypes*. By default, types refer to the more general form, *i.e.* to 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 σ' where all occurrences of α would have been replaced by the polytype σ. However, our notation contains additional meaningful sharing information. Finally, the general form $\forall(\alpha \geq \sigma)\ \sigma'$ intuitively stands for the collection of *all polytypes* σ' *where* α *is an instance of* σ.

Notation. We say that α 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 (α). For convenience, we write $(\alpha \diamond \sigma)$ for either $(\alpha = \sigma)$ or $(\alpha \geq \sigma)$. The symbol $\diamond$ 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 and free variables. Polytypes are considered equal modulo α-conversion where $\forall(\alpha \diamond \sigma)\ \sigma'$ binds α in σ', but not in σ. The set of free variables of a polytype σ is written $\mathsf{ftv}(\sigma)$ and defined inductively as follows:

$$\mathsf{ftv}(\alpha) = \{\alpha\} \qquad \mathsf{ftv}(g^n\ \tau_1\ \ldots\ \tau_n) = \bigcup_{i=1..n} \mathsf{ftv}(\tau_i) \qquad \mathsf{ftv}(\bot) = \emptyset$$

$$\mathsf{ftv}(\forall(\alpha \diamond \sigma)\ \sigma') = \begin{cases} \mathsf{ftv}(\sigma') & \text{if } \alpha \notin \mathsf{ftv}(\sigma') \\ \mathsf{ftv}(\sigma') \setminus \{\alpha\} \cup \mathsf{ftv}(\sigma) & \text{otherwise} \end{cases}$$

The capture-avoiding substitution of α by τ in σ is written $\sigma[\tau/\alpha]$.

EXAMPLE 1. The syntax of types only allows quantifiers to 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 τ in which the variable β 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— an encoding from System F into ML^{F} is given in Section 5.1.

Types may be instantiated implicitly as in ML along an instance relation $\preccurlyeq$. As explained above, we decompose $\preccurlyeq$ into $\exists\sqsubseteq\exists$. In

---

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

Section 1.2, we first define an equivalence relation between types, which is the kernel of both $\mathrel{\sqsubseteq\!\!\!\!-}$ and $\sqsubseteq$. In Section 1.3, we define the relation $\mathrel{\sqsubseteq\!\!\!\!-}$ that is the inverse of $\mathrel{-\!\!\!\!\sqsupseteq}$. The instance relation $\sqsubseteq$, which contains $\mathrel{\sqsubseteq\!\!\!\!-}$, is defined in Section 1.4.

1.2 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 and all other relations between types are relative to a prefix that specifies the bounds of free type variables.

DEFINITION 1 (PREFIXES). A *prefix* 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 and form the domain of Q, which we write $\mathsf{dom}(Q)$. The order of bindings in a prefix is significant: bindings are meant to be read from left to right; furthermore, we require that variables α_j do not occur free in σ_i whenever $i \le j$. 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 $(Q_1, \alpha\diamond\sigma, Q_2)$. We also write $\forall(Q)\ \sigma$ for the type $\forall(\alpha_1 \diamond_1 \sigma_1)\ldots\forall(\alpha_n \diamond_n \sigma_n)\ \sigma$. (Note that α_i's can be renamed in the type $\forall(Q)\ \sigma$, but not in the prefix Q.) ☐

DEFINITION 2 (EQUIVALENCE). The *equivalence under prefix* is a relation on triples composed of a prefix Q and two types σ_1 and σ_2, written $(Q)\ \sigma_1 \equiv \sigma_2$. It is defined as the smallest relation that satisfies the rules of Figure 2. We write $\sigma_1 \equiv \sigma_2$ for $(\emptyset)\ \sigma_1 \equiv \sigma_2$. ☐

Rule EQ-COMM allows the reordering of independent binders; Rule EQ-FREE eliminates unused bound variables. Rules EQ-CONTEXT-L and EQ-CONTEXT-R tell that $\equiv$ is a congruence; Reasoning under prefixes allows to break up a polytype $\forall(Q)\ \sigma$ 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 $(\emptyset)\ \forall(Q)\ \sigma \equiv \forall(Q)\ \sigma'$.

Rule EQ-MONO allows to *read* the bound of a variable from the prefix when it is (equivalent to) a monotype. An example of use of EQ-MONO is $(Q, \alpha = \tau_0, Q')\ \alpha \to \alpha \equiv \tau_0 \to \tau_0$. Rule EQ-MONO makes no difference between $\ge$ and $=$ whenever the bound is (equivalent to) a monotype. The restriction of Rule EQ-MONO to the case where σ_0 is (equivalent to) a monotype is required for the well-formedness of the conclusion. Moreover, it also disallows $(Q, \alpha = \sigma_0, Q')\ \alpha \equiv \sigma_0$ when τ is a variable α: variables with non trivial bounds must be treated abstractly and cannot be silently expanded. In particular, $(Q)\ \forall(\alpha = \sigma_0, \alpha' = \sigma_0)\ \alpha \to \alpha' \equiv \forall(\alpha = \sigma_0)\ \alpha \to \alpha$ does not hold.

Rule EQ-VAR expands into both $\forall(\alpha = \sigma)\ \alpha \equiv \sigma$ and $\forall(\alpha \ge \sigma)\ \alpha \equiv \sigma$. The former captures the intuition that $\forall(\alpha = \sigma)\ \sigma'$ stands for $\sigma'[\sigma/\alpha]$, which however, is not always well-formed. The latter may be surprising, since one could expect $\forall(\alpha \ge \sigma)\ \alpha \sqsubseteq \sigma$ to hold, but not the converse. The inverse part of the equivalence could be removed without changing the set of typable terms. However, it is harmless and allows for a more uniform presentation.

The equivalence $(Q)\ \forall(\alpha\diamond\tau)\ \sigma \equiv \sigma[\tau/\alpha]$ follows from Rules EQ-MONO, context rules, transitivity, and EQ-FREE, which we further refer to as the derived rule EQ-MONO*.

The equivalence under (a given) prefix is a symmetric operation. In other words, it captures reversible transformations. Irreversible transformations are captured by an *instance* relation $\sqsubseteq$. Moreover, we distinguish a subrelation $\mathrel{\sqsubseteq\!\!\!\!-}$ of $\sqsubseteq$ called *abstraction*. Inverse of abstractions are sound relations for $\preccurlyeq$ but made explicit so as to preserve type inference, while inverse of instance relations would, in general, be unsound for $\preccurlyeq$.

Figure 2. Type equivalence under prefix

All rules are considered symmetrically.

EQ-REFL
$$(Q)\ \sigma \equiv \sigma$$

EQ-TRANS
$$\frac{(Q)\ \sigma_1 \equiv \sigma_2 \qquad (Q)\ \sigma_2 \equiv \sigma_3}{(Q)\ \sigma_1 \equiv \sigma_3}$$

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

Figure 3. The abstraction relation

A-EQUIV
$$\frac{(Q)\ \sigma_1 \equiv \sigma_2}{(Q)\ \sigma_1 \mathrel{\sqsubseteq\!\!\!\!-} \sigma_2}$$

A-TRANS
$$\frac{(Q)\ \sigma_1 \mathrel{\sqsubseteq\!\!\!\!-} \sigma_2 \qquad (Q)\ \sigma_2 \mathrel{\sqsubseteq\!\!\!\!-} \sigma_3}{(Q)\ \sigma_1 \mathrel{\sqsubseteq\!\!\!\!-} \sigma_3}$$

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

A-HYP
$$\frac{(\alpha_1 = \sigma_1) \in Q}{(Q)\ \sigma_1 \mathrel{\sqsubseteq\!\!\!\!-} \alpha_1}$$

A-CONTEXT-L
$$\frac{(Q)\ \sigma_1 \mathrel{\sqsubseteq\!\!\!\!-} \sigma_2}{(Q)\ \forall(\alpha = \sigma_1)\ \sigma \mathrel{\sqsubseteq\!\!\!\!-} \forall(\alpha = \sigma_2)\ \sigma}$$

1.3 The abstraction relation

DEFINITION 3. The *abstraction under prefix*, is a relation on triples composed of a prefix Q and two types σ_1 and σ_2, written[3] $(Q)\ \sigma_1 \mathrel{\sqsubseteq\!\!\!\!-} \sigma_2$, and defined as the smallest relation that satisfies the rules of Figure 3. We write $\sigma_1 \mathrel{\sqsubseteq\!\!\!\!-} \sigma_2$ for $(\emptyset)\ \sigma_1 \mathrel{\sqsubseteq\!\!\!\!-} \sigma_2$. ☐

Rules A-CONTEXT-L and A-CONTEXT-R are context rules; note that Rule A-CONTEXT-L does not allow abstraction under flexible bounds. The interesting rule is A-HYP, which replaces a polytype σ_1 by a variable α_1, provided α_1 is rigidly bound to σ_1 in Q.

Remarkably, rule A-HYP is not reversible. In particular, $(\alpha = \sigma) \in Q$ does not imply $(Q)\ \alpha \mathrel{\sqsubseteq\!\!\!\!-} \sigma$, unless σ is (equivalent to) a monotype. This asymmetry is essential, since uses of $\mathrel{\sqsubseteq\!\!\!\!-}$ will be inferred, but uses of $\mathrel{-\!\!\!\!\sqsupseteq}$ will not. Intuitively, the former consists in *abstracting* the polytype σ as the name α (after checking that α is declared as an alias for σ in Q). The latter consists in *revealing* the polytype abstracted by the name α. An abstract polytype, *i.e.* a variable bound to a polytype in Q, can only be manipulated by its name, *i.e.* abstractly. The polytype must be revealed *explicitly* (by using the relation $\mathrel{-\!\!\!\!\sqsupseteq}$) before it can be further instantiated (along the relation $\mathrel{\sqsubseteq\!\!\!\!-}$ or $\sqsubseteq$). (See also examples 6 and 7.)

---

[3] Read σ_2 is an abstraction of σ_1—or σ_1 is a revelation of σ_2—under prefix Q.

Figure 4. Type instance

$$\textsc{I-Abstract}\quad \frac{(Q)\ \sigma_1 \mathrel{-\!\!\!\!\sqsubseteq} \sigma_2}{(Q)\ \sigma_1 \sqsubseteq \sigma_2} \qquad \textsc{I-Trans}\quad \frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2 \quad (Q)\ \sigma_2 \sqsubseteq \sigma_3}{(Q)\ \sigma_1 \sqsubseteq \sigma_3} \qquad \textsc{I-Context-R}\quad \frac{(Q, \alpha \diamond \sigma)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \forall(\alpha \diamond \sigma)\ \sigma_1 \sqsubseteq \forall(\alpha \diamond \sigma)\ \sigma_2}$$

$$\textsc{I-Hyp}\quad \frac{(\alpha_1 \geq \sigma_1) \in Q}{(Q)\ \sigma_1 \sqsubseteq \alpha_1} \qquad \textsc{I-Context-L}\quad \frac{(Q)\ \sigma_1 \sqsubseteq \sigma_2}{(Q)\ \forall(\alpha \geq \sigma_1)\ \sigma \sqsubseteq \forall(\alpha \geq \sigma_2)\ \sigma}$$

$$\textsc{I-Bot}\quad (Q)\ \bot \sqsubseteq \sigma \qquad \textsc{I-Rigid}\quad \frac{}{(Q)\ \forall(\alpha \geq \sigma_1)\ \sigma \sqsubseteq \forall(\alpha = \sigma_1)\ \sigma}$$

Example 2. The abstraction $(\alpha = \sigma)\ \forall(\alpha = \sigma)\ \sigma' \mathrel{-\!\!\!\!\sqsubseteq} \sigma'$ is derivable: on the one hand, $(\alpha = \sigma)\ \sigma \mathrel{-\!\!\!\!\sqsubseteq} \alpha$ holds by A-Hyp, leading to $(\alpha = \sigma)\ \forall(\alpha = \sigma)\ \sigma' \mathrel{-\!\!\!\!\sqsubseteq} \forall(\alpha = \alpha)\ \sigma'$ by A-Context-L; on the other hand, $(\alpha = \sigma)\ \forall(\alpha = \alpha)\ \sigma' \equiv \sigma'$ holds by Eq-Mono*. Hence, we conclude by A-Equiv and A-Trans.

1.4 The instance relation

Definition 4. The *instance under prefix*, is a relation on triples composed of a prefix Q and two types σ_1 and σ_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 $(\emptyset)\ \sigma_1 \sqsubseteq \sigma_2$. ☐

Rule I-Bot means that $\bot$ behaves as a 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 it is an abstraction, as described by Rule A-Context-L.

The interesting rule is I-Hyp—the counter-part of rule A-Hyp, which replaces a polytype σ_1 by a variable α_1, provided σ_1 is a flexible bound of α_1 in Q.

Example 3. The instance relation $(\alpha \geq \sigma)\ \forall(\alpha \geq \sigma)\ \sigma' \sqsubseteq \sigma'$ holds. The derivation follows the one of Example 2 but uses I-Hyp and I-Context-L instead of A-Hyp and A-Context-L. More generally, $(QQ')\ \forall(Q')\ \sigma \sqsubseteq \sigma$ holds for any Q, Q', and σ, which we refer to as Rule I-Drop.

The relation $(Q)\ \forall(\alpha_1 \geq \forall(\alpha_2 \diamond \sigma_2)\ \sigma_1)\ \sigma \sqsubseteq \forall(\alpha_2 \diamond \sigma_2)\ \forall(\alpha_1 \geq \sigma_1)\ \sigma$ holds whenever $\alpha_2 \notin \mathsf{ftv}(\sigma)$, which we further refer to as the derived rule I-Up.

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

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

The instance relation coincide with equivalence on monotypes, which captures the intuition that "monotypes are really monomorphic".

Lemma 2. *For all prefixes Q and monotypes τ and τ', we have $(Q)\ \tau \sqsubseteq \tau'$ if and only if $(Q)\ \tau \equiv \tau'$.*

The instance relation also coincides with the one of ML on ML-types. In particular, $\forall(\bar{\alpha})\ \tau_0 \sqsubseteq \tau_1$ if and only if τ_1 is of the form $\tau_0[\bar{\tau}/\bar{\alpha}]$.

Example 4. The instance relation covers an interesting case of type isomorphism [3]. In System F, type $\forall\alpha \cdot \tau' \to \tau$ is isomorphic[5] to $\tau' \to \forall\alpha \cdot \tau$ whenever α is not free in τ'. In ML$^{\mathrm{F}}$, the two corresponding polytypes 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:

$$\begin{array}{ll} \ \forall(\alpha' \geq \forall(\alpha)\ \tau)\ \tau' \to \alpha' & \\ \sqsubseteq \forall(\alpha)\ \forall(\alpha' \geq \tau)\ \tau' \to \alpha' & \text{by I-Up} \\ \equiv \forall(\alpha)\ \tau' \to \tau & \text{by Eq-Mono}^\star \end{array}$$

(However, as opposed to type containment [19], the instance relation cannot express any form of contravariance.)

1.5 Operation on prefixes and unification

Rules A-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, for any suffix σ. 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 lists (a superset of) the free type variables of σ. In this context, a set of type variables is called an *interface* and is written with letter I.

Definition 5 (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 types σ 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 $Q \mathrel{-\!\!\!\!\sqsubseteq}^I Q'$ similarly. ☐

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'.

Definition 6 (Unification). A prefix Q' *unifies* monotypes τ_1 and τ_2 under Q if and only if $Q \sqsubseteq Q'$ and $(Q)\ \tau_1 \equiv \tau_2$. ☐

The unification algorithm, called `unify`, is defined in Appendix A.

Theorem 1. *For any prefix Q and monotypes τ_1 and τ_2,* `unify` *(Q, τ_1, τ_2) returns the smallest prefix (for the relation $\sqsubseteq^{\mathsf{dom}(Q)}$) that unifies τ_1 and τ_2 under Q, or fails if there exists no prefix Q' that unifies τ_1 and τ_2 under Q.*

The following lemma shows that first-order unification lies under ML$^{\mathrm{F}}$ unification.

Lemma 3. *If $(Q)\ \tau_1 \equiv \tau_2$, then there exists a substitution $\widehat{Q}$ (depending only on Q) that unifies τ_1 and τ_2.*

2 The core language

As explained in the introduction we formalize the language ML$^{\mathrm{F}}$ as a restriction to the more permissive language ML$^{\mathrm{F}}_{\star}$. We assume given a countable set of variables, written with letter x, and a countable set of constants $c \in \mathcal{C}$. Every constant c has an arity $|c|$. A constant is either a primitive f or a constructor C. The distinction

---

[4] Read σ_2 *is an instance of* σ_1—or σ_1 *is more general than* σ_2—*under prefix Q.*

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

Figure 5. Expressions of $\text{ML}^{\text{F}}_{\star}$

$a ::= x \mid c \mid \lambda(x)\ a \mid a\ a \mid \texttt{let}\ x = a\ \texttt{in}\ a$	Terms
$\quad \mid (a : \star)$	Oracles
$c ::= f \mid C$	Constants
$z ::= x \mid c$	Identifiers

between constructors and primitives lies in their dynamic semantics: primitives (such as $+$) are reduced when fully applied, while constructors (such as `cons`) represent data structures, and are not reduced. We use letter z to refer to identifiers, *i.e.* either variables or constants.

Expressions of $\text{ML}^{\text{F}}_{\star}$, written with letter a, are described in Figure 5. Expressions are those of ML extended with *oracles*. An oracle, written $(a : \star)$ is simply a place holder for an implicit type annotation around the expression a. Intuitively, oracles are places where the type inference algorithm must call an "oracle" to fill the hole with a type annotation. Equivalently, the oracles can be replaced by explicit type annotations before type inference. Explicit annotations $(a : \sigma)$, which are described in Section 4, are actually syntactic sugar for applications $(\sigma)\ a$ where (σ) are constants. Examples in the introduction also use the notation $\lambda(x : \sigma)\ a$, which do not appear in Figure 5, because this is, again, syntactic sugar for $\lambda(x)\ \texttt{let}\ x = (x : \sigma)\ \texttt{in}\ a$. Similarly, $\lambda(x : \star)\ a$ means $\lambda(x)\ \texttt{let}\ x = (x : \star)\ \texttt{in}\ a$.

The language ML^{F} is the restriction of $\text{ML}^{\text{F}}_{\star}$ to expressions that do not contain oracles.

2.1 Static semantics

Typing contexts, written with letter Γ are lists of assertions of the form $z : \sigma$. We write $z : \sigma \in \Gamma$ to mean that z is bound in Γ and $z : \sigma$ is its rightmost binding in Γ. We assume given an initial typing context Γ_0 mapping constants to closed polytypes.

Typing judgments are of the form $(Q)\ \Gamma \vdash a : \sigma$. A tiny difference with ML is the presence of the prefix Q that assigns bounds to type variables appearing free in Γ or σ. By comparison, this prefix is left implicit in ML because all free type variables have the same (implicit) bound $\bot$. In ML^{F}, we require that σ and all polytypes of Γ 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 $\text{ML}^{\text{F}}_{\star}$ and 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 type variables in judgments. In addition, Rule ORACLE allows for the *revelation* of polytypes, that is, the transformation of types along the inverse of the abstraction relation. (This rule would has no effect in ML where abstraction is the same as equivalence.) For UML^{F}, it suffices to replace Rule ORACLE by U-ORACLE given below or, equivalently, combine ORACLE with INST into U-INST.

U-ORACLE
$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \sigma \exists \sigma'}{(Q)\ \Gamma \vdash a : \sigma'}$$

U-INST
$$\frac{(Q)\ \Gamma \vdash a : \sigma \qquad (Q)\ \sigma \exists\sqsubseteq\exists\ \sigma'}{(Q)\ \Gamma \vdash a : \sigma'}$$

As in ML, there is an important difference between rules FUN and LET: while typechecking their bodies, a let-bound variable can be assigned a polytype, but a λ-bound variable can only be assigned a monotype in Γ. 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. Notice that a λ-bound variable can refer to a polytype *abstractly* via a type variable α bound to a polytype σ in Q. However, this will not allow to take different instances of σ while typing the body of the abstraction, unless the polytype bound σ of α is first *revealed* by an oracle. Indeed, the only possible instances of α under a prefix Q that contains the binding $(\alpha = \sigma)$ are types equivalent to α under Q. However, $(Q)\ \alpha \equiv \sigma$ does not hold. Thus, if $x : \alpha$ is in the typing context Γ, the only way of typing x (modulo equivalence) is $(Q)\ \Gamma \vdash x : \alpha$, whereas $(Q)\ \Gamma \vdash x : \sigma$ is not derivable. Conversely, $(Q)\ \Gamma \vdash (x : \star) : \sigma$ is derivable, since $(Q)\ \alpha \exists \sigma$.

Figure 6. Typing rules for ML^{F} and $\text{ML}^{\text{F}}_{\star}$

VAR
$$\frac{z : \sigma \in \Gamma}{(Q)\ \Gamma \vdash z : \sigma}$$

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

FUN
$$\frac{(Q)\ \Gamma, x : \tau_0 \vdash a : \tau}{(Q)\ \Gamma \vdash \lambda(x)\ a : \tau_0 \to \tau}$$

LET
$$\frac{\begin{array}{c}(Q)\ \Gamma \vdash a_1 : \sigma \\ (Q)\ \Gamma, x : \sigma \vdash a_2 : \tau\end{array}}{(Q)\ \Gamma \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau}$$

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

INST
$$\frac{\begin{array}{c}(Q)\ \Gamma \vdash a : \sigma \\ (Q)\ \sigma \sqsubseteq \sigma'\end{array}}{(Q)\ \Gamma \vdash a : \sigma'}$$

ORACLE
$$\frac{\begin{array}{c}(Q)\ \Gamma \vdash a : \sigma \\ (Q)\ \sigma \exists \sigma'\end{array}}{(Q)\ \Gamma \vdash (a : \star) : \sigma'}$$

ML as a subset of ML^{F}. ML can be embedded into ML^{F} by restricting all bounds in the prefix Q to be unconstrained. Rules GEN and INST are then exactly those of ML. Hence, any closed program typable in ML is also typable in ML^{F}.

EXAMPLE 5. This first example of typing illustrates the use of polytypes in typing derivations: we consider the simple expression K' defined by $\lambda(x)\ \lambda(y)\ y$. Following ML, one possible typing derivation is (we recall that (α, β) stands for $(\alpha \geq \bot, \beta \geq \bot)$):

$$\text{GEN}\ \dfrac{\text{FUN}\ \dfrac{\text{FUN}\ \dfrac{(\alpha, \beta)\ x : \alpha, y : \beta \vdash y : \beta}{(\alpha, \beta)\ x : \alpha \vdash \lambda(y)\ y : \beta \to \beta}}{(\alpha, \beta)\ \vdash K' : \alpha \to (\beta \to \beta)}}{\vdash K' : \forall(\alpha, \beta)\ \alpha \to (\beta \to \beta)}$$

There is, however, another typing derivation that infers a more general type for K' in ML^{F} (for conciseness we write Q for $\alpha, \beta \geq \sigma_{\texttt{id}}$):

$$\text{GEN}\ \dfrac{\text{FUN}\ \dfrac{\text{INST}\ \dfrac{\text{GEN}\ \dfrac{\text{FUN}\ \dfrac{(Q, \gamma)\ x : \alpha, y : \gamma \vdash y : \gamma}{(Q, \gamma)\ x : \alpha \vdash \lambda(y)\ y : \gamma \to \gamma}}{(Q)\ x : \alpha \vdash \lambda(y)\ y : \sigma_{\texttt{id}}} \qquad (Q)\ \sigma_{\texttt{id}} \sqsubseteq \beta}{(Q)\ x : \alpha \vdash \lambda(y)\ y : \beta}}{(Q)\ \vdash K' : \alpha \to \beta}}{\vdash K' : \forall(Q)\ \alpha \to \beta}$$

Notice that the polytype $\forall(\alpha, \beta \geq \sigma_{\texttt{id}})\ \alpha \to \beta$ is more general than $\forall(\alpha, \beta)\ \alpha \to (\beta \to \beta)$, which follows from Example 4.

Figure 7. Syntax directed typing rules

$$\textsc{Var}^\triangledown\ \dfrac{z : \sigma \in \Gamma}{(Q)\ \Gamma \vdash^\triangledown z : \sigma} \qquad \textsc{Fun}^\triangledown\ \dfrac{(QQ')\ \Gamma, x : \tau_0 \vdash^\triangledown a : \sigma \qquad \mathsf{dom}(Q') \cap \Gamma = \emptyset}{(Q)\ \Gamma \vdash^\triangledown \lambda(x)\ a : \forall(Q', \alpha \geq \sigma)\ \tau_0 \to \alpha}$$

$$\textsc{App}^\triangledown\ \dfrac{\begin{array}{cc} (Q)\ \Gamma \vdash^\triangledown a_1 : \sigma_1 & (Q)\ \Gamma \vdash^\triangledown a_2 : \sigma_2 \\ (Q)\ \sigma_1 \sqsubseteq \forall(Q')\ \tau_2 \to \tau_1 & (Q)\ \sigma_2 \sqsubseteq \forall(Q')\ \tau_2 \end{array}}{(Q)\ \Gamma \vdash^\triangledown a_1\ a_2 : \forall(Q')\ \tau_1}$$

$$\textsc{Let}^\triangledown\ \dfrac{\begin{array}{c} (Q)\ \Gamma \vdash^\triangledown a_1 : \sigma_1 \\ (Q)\ \Gamma, x : \sigma_1 \vdash^\triangledown a_2 : \sigma_2 \end{array}}{(Q)\ \Gamma \vdash^\triangledown \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \sigma_2} \qquad \textsc{Oracle}^\triangledown\ \dfrac{\begin{array}{c} (Q)\ \Gamma \vdash^\triangledown a : \sigma \\ (Q)\ \sigma \mathrel{\exists} \sigma' \end{array}}{(Q)\ \Gamma \vdash^\triangledown (a : \star) : \sigma'}$$

2.2 Syntax directed presentation

As in ML, we can replace the typing rules of $\mathrm{ML}^{\mathrm{F}}_{\star}$ by a set of equivalent syntax-directed typing rules, which are given in Figure 7. Naively, a sequence of non-syntax-directed typing Rules Gen and Inst should be placed around any other rule. However, many of these occurrences can be proved unnecessary by following an appropriate strategy. For instance, in ML, judgments are maintained instantiated as much as possible and are only generalized on the left-hand side of Rule Let. In $\mathrm{ML}^{\mathrm{F}}_{\star}$, this strategy would require more occurrences of generalization. Instead, we prefer to maintain typing judgments generalized as much as possible. Then, it suffices to allow Rule Gen right after Rule Fun and to allow Rule Inst right before Rule App (see Rules $\textsc{Fun}^\triangledown$ and $\textsc{App}^\triangledown$).

Example 6. As we claimed in the introduction, a λ-bound variable that is used polymorphically must be annotated. Let us check that $\lambda(x)\ x\,x$ is not typable in ML^{F} by means of contradiction. A syntax-directed type derivation of this expression would be of the form:

$$\textsc{Fun}^\triangledown\ \dfrac{\textsc{App}^\triangledown\ \dfrac{\begin{array}{cc} \textsc{Var}^\triangledown\ (Q)\ x : \tau_0 \vdash^\triangledown x : \tau_0 & (Q)\ x : \tau_0 \vdash^\triangledown x : \tau_0\ \textsc{Var}^\triangledown \\ (Q)\ \tau_0 \sqsubseteq \forall(Q')\ \tau_2 \to \tau_1\ (\mathbf{2}) & (Q)\ \tau_0 \sqsubseteq \forall(Q')\ \tau_2\ (\mathbf{1}) \end{array}}{(Q)\ x : \tau_0 \vdash^\triangledown x\,x : \forall(Q')\ \tau_1}}{(Q)\ \emptyset \vdash \lambda(x)\ x\,x : \forall(\alpha \geq \forall(Q')\ \tau_1)\ \tau_0 \to \alpha}$$

Applying Rule I-Drop to (2) and (1), we get respectively $(QQ')\ \tau_0 \sqsubseteq \tau_2 \to \tau_1$ and $(QQ')\ \tau_0 \sqsubseteq \tau_2$. Then $(QQ')\ \tau_2 \to \tau_1 \equiv \tau_2$ follows by Lemma 2 and Eq-Trans. Thus, by Lemma 3, there exists a substitution θ such that $\theta(\tau_2) = \theta(\tau_2 \to \tau_1)$, that is, $\theta(\tau_2) = \theta(\tau_2) \to \theta(\tau_1)$, which cannot be the case.

This example shows the limit of type inference, which is actually the strength of our system! That is, to maintain principal types by rejecting examples where type inference would need to guess second-order types.

Example 7. Let us recover typability by introducing an oracle and build a derivation for $\lambda(x)\ (x : \star)\ x$. Taking $(\alpha = \sigma_{\mathtt{id}})$ for Q and α for τ_0, we obtain:

$$\textsc{Fun}^\triangledown\ \dfrac{\textsc{App}^\triangledown\ \dfrac{\textsc{Oracle}^\triangledown\ \dfrac{\begin{array}{c} \textsc{Var}^\triangledown\ (Q)\ x : \alpha \vdash^\triangledown x : \alpha \\ (Q)\ \alpha \mathrel{\exists} \sigma_{\mathtt{id}}\ (\mathbf{3}) \end{array}}{(Q)\ x : \alpha \vdash^\triangledown (x : \star) : \sigma_{\mathtt{id}}} \qquad \begin{array}{c} (Q)\ \sigma_{\mathtt{id}} \sqsubseteq \alpha \to \alpha \\ (Q)\ x : \alpha \vdash^\triangledown x : \alpha\ \textsc{Var}^\triangledown \end{array}}{(Q)\ x : \alpha \vdash^\triangledown (x : \star)\ x : \alpha}}{\vdash \lambda(x)\ (x : \star)\ x : \forall(\alpha = \sigma_{\mathtt{id}})\ \alpha \to \alpha}$$

The oracle plays a crucial role in (3)—the revelation of the type scheme $\sigma_{\mathtt{id}}$ that is the bound of the type variable α used in the type of x. We have $(Q)\ \sigma_{\mathtt{id}} \sqsubseteq \alpha$, indeed, but the converse relation does not hold, so rule Inst cannot be used here to replace α by its bound $\sigma_{\mathtt{id}}$.

2.3 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.

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

The reduction relation $\longrightarrow$ is parameterized by a set of δ-rules of the form (δ) below:

$$\begin{array}{rcll} f\ v_1\ \ldots\ v_n & \longrightarrow & a \quad \text{when } |f| = n & (\delta) \\ (\lambda(x)\ a)\ v & \longrightarrow & a[v/x] & (\beta_v) \\ \mathtt{let}\ x = v\ \mathtt{in}\ a & \longrightarrow & a[v/x] & (\beta_{let}) \\ (v_1 : \star)\ v_2 & \longrightarrow & (v_1\ v_2 : \star) & (\star) \end{array}$$

The main reduction is the β-reduction that takes two forms Rule (β_v) and Rule (β_{let}). Oracles are maintained during reduction to which they do not contribute: they are simply pushed out of applications by rule ($\star$). Finally, the reduction is the smallest relation containing (δ), (β_v), (β_{let}), and ($\star$) rules that is closed by E-congruence:

$$E[a] \longrightarrow E[a'] \text{ if } a \longrightarrow a' \qquad (\textsc{Context})$$

3 Formal properties

We verify type soundness for $\mathrm{ML}^{\mathrm{F}}_{\star}$ and address type inference in ML^{F}.

3.1 Type soundness

Type soundness for $\mathrm{ML}^{\mathrm{F}}_{\star}$ is shown as usual 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, Γ, σ) 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 relating the static semantics of constants described by the initial typing context Γ_0 and their dynamic semantics.

DEFINITION 7 (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 $\forall (Q)\ \tau$ is not in $\{\to, \bot\}$ whenever c is a constructor.

(H1) (Subject-Reduction) All δ-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 (δ). □

THEOREM 2 (SUBJECT REDUCTION). *Reduction preserves typings.*

THEOREM 3 (PROGRESS). *Any expression a such that* $(Q)\ \Gamma_0 \vdash a : \sigma$ *is a value or can be further reduced.*

Combining theorems 2 and 3 ensures that the reduction of well-typed programs either proceeds for ever or ends up with a value. This holds for programs in $\mathrm{ML}^{\mathrm{F}}_{\star}$ but also for programs in ML^{F}, since ML^{F} is a subset of $\mathrm{ML}^{\mathrm{F}}_{\star}$. Hence ML^{F} is also sound. However, ML^{F} does not enjoy subject reduction, since reduction may create oracles. Notice, however, that oracles can only be introduced by δ-rules.

3.2 Type inference

A *type inference problem* is a triple (Q, Γ, a), where all free type variables in Γ are bound in Q. A pair (Q', σ) is a *solution* to this problem if $Q \sqsubseteq Q'$ and $(Q')\ \Gamma \vdash a : \sigma$. A pair (Q', σ') is an *instance* of a pair (Q, σ) 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 B defines a type inference algorithm $\mathcal{W}^{\mathrm{F}}$ for ML^{F}. This algorithm proceeds much as type inference for ML: the algorithm follows the syntax-directed typing rules and reduces type inference to unification under prefixes.

THEOREM 4 (TYPE INFERENCE). *The set of solutions of a solvable type inference problem admits a principal solution. Given any type inference problem, the algorithm* $\mathcal{W}^{\mathrm{F}}$ *either returns a principal solution or fails if no solution exists.*

4 Type annotations

In this section, we restrict our attention to ML^{F}, *i.e.* to expressions that do not contain oracles. Since expressions of ML^{F} are exactly those of ML, its expressiveness may only come from richer types and typing rules. However, the following lemma shows that this is not sufficient:

LEMMA 4. *If the judgment* $(Q)\ \Gamma \vdash a : \sigma$ *holds in* ML^{F} *where the typing context* Γ *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* σ *by moving all inner quantifiers ahead.*

The inverse inclusion has already been stated in Section 2.1. In the particular case where the initial typing context Γ_0 contains only ML types, a closed expression can be typed in ML^{F} under Γ_0 if and only if it can be typed in ML under Γ_0. This is not true for $\mathrm{ML}^{\mathrm{F}}_{\star}$ in which the expression $\lambda(x)\ (x : \star)\ x$ is typable. Indeed, as shown below, all terms of System F can be typed in $\mathrm{ML}^{\mathrm{F}}_{\star}$. Fortunately, there is an interesting choice of constants that provides ML^{F} with the same expressiveness as $\mathrm{ML}^{\mathrm{F}}_{\star}$ 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 a single annotation, should provide intuition for the general case.

EXAMPLE 8. Let f be a constant of type σ equal to $\forall (\alpha = \sigma_{\mathtt{id}}, \alpha' \geq \sigma_{\mathtt{id}})\ \alpha \to \alpha'$ with the δ-reduction $f\ v \longrightarrow (v : \star)$. Then, the expression a defined as $\lambda(x)\ (f\ x)\ x$ behaves as $\lambda(x)\ x\ x$ and is well-typed, of type $\forall (\alpha = \sigma_{\mathtt{id}})\ \alpha \to \alpha$. To see this, let Q and Γ stand for $(\alpha = \sigma_{\mathtt{id}}, \alpha' = \sigma_{\mathtt{id}})$ and $x : \alpha$. By Rules INST, VAR, and APP $(Q)\ \Gamma \vdash f\ x : \alpha'$; hence by rule GEN, $(\alpha = \sigma_{\mathtt{id}})\ \Gamma \vdash f\ x : \forall (\alpha' = \sigma_{\mathtt{id}})\ \alpha'$ since α' is not free in the Γ. By rule EQ-VAR, we have $\forall (\alpha' = \sigma_{\mathtt{id}})\ \alpha' \equiv \sigma_{\mathtt{id}}$ (under any prefix); besides, $\sigma_{\mathtt{id}} \sqsubseteq \alpha \to \alpha$ under any prefix that binds α. Thus, we get $(\alpha = \sigma_{\mathtt{id}})\ \Gamma \vdash f\ x : \alpha \to \alpha$ by Rule INST. The result follows by Rules APP, FUN, and GEN.

Observe that the static effect of f in $f\ x$ is (i) to enforce the type of x to be abstracted by a variable α bound to $\sigma_{\mathtt{id}}$ in Q and (ii) to give $f\ x$, that is x, the type $\sigma_{\mathtt{id}}$, exactly as the oracle $(x : \star)$ would. Notice that the bound of α in σ is rigid: the function f expects a value v that must have type $\sigma_{\mathtt{id}}$ (and not a strict instance of $\sigma_{\mathtt{id}}$). Conversely, the bound of α' is flexible: the type of $f\ v$ is $\sigma_{\mathtt{id}}$ but may also be any instance of $\sigma_{\mathtt{id}}$.

DEFINITION 8. We call *annotations* the denumerable collection of primitives $(\exists (Q)\ \sigma)$, of arity 1, defined for all prefixes Q and polytypes σ closed under Q. The initial typing environment Γ_0 contains these primitives with their associated type:

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

We may identify annotation primitives up to the equivalence of their types. □

Besides, we write $(a : \exists (Q)\ \sigma)$ for the application $(\exists (Q)\ \sigma)\ a$. We also abbreviate $(\exists (Q)\ \sigma)$ as (σ) when all bounds in Q are unconstrained. Actually, replacing an annotation $(\exists (Q)\ \sigma)$ by (σ) preserves typability and, more precisely, preserves typings.

While annotations are introduced as primitives for simplicity of presentation, they are obviously meant to be applied. Notice that the type of an annotation may be instantiated before the annotation is applied. However, the annotation keeps exactly the same “revealing power” after instantiation. This is described by the following technical lemma (the reader may take $\emptyset$ for Q_0 at first).

LEMMA 5. *The judgment* $(Q_0)\ \Gamma \vdash (a : \exists (Q)\ \sigma) : \sigma_0$ *is valid if and only if there exists a type* $\forall (Q')\ \sigma'_1$ *such that the judgment* $(Q_0)\ \Gamma \vdash a : \forall (Q')\ \sigma'_1$ *holds together with the following relations:* $Q_0 Q \sqsubseteq Q_0 Q'$, $(Q_0 Q')\ \sigma'_1 \exists \sigma$, *and* $(Q_0)\ \forall (Q')\ \sigma \sqsubseteq \sigma_0$.

The prefix Q of the annotation $\exists (Q)\ \sigma$ may be instantiated into Q'. However, Q' guards $\sigma'_1 \exists \sigma$ in $(Q_0 Q')\ \sigma'_1 \exists \sigma$. In particular, the lemma would not hold with $(Q_0)\ \forall (Q')\ \sigma'_1 \exists \forall (Q'')\ \sigma$ and $(Q_0)\ \forall (Q'')\ \sigma'_1 \sqsubseteq \sigma_0$. Lemma 5 has similarities with Rule ANNOT of Poly-ML [7].

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

Hence, all expressions typable in $\mathrm{ML}^{\mathrm{F}}_{\star}$ are typable in ML^{F} as long as all annotation primitives are in the initial typing context Γ_0.

Reduction of annotations. The δ-reduction for annotations just replaces explicit type information by oracles.

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

LEMMA 7 (SOUNDNESS OF TYPE ANNOTATIONS). *All three hypotheses (H0, arity), (H1, subject-reduction), and (H2, progress) hold when primitives are the set of annotations, alone.*

The annotation $(\exists\,(Q)\ \sigma)$ can be simulated by $\lambda(x)\ (x : \star)$ in $\mathrm{ML}^{\mathrm{F}}_{\star}$, both statically and dynamically. Hence annotations primitives are unnecessary in $\mathrm{ML}^{\mathrm{F}}_{\star}$.

Syntactic sugar. As mentioned in Section 2, $\lambda(x : \sigma)\ a$ is syntactic sugar for $\lambda(x)$ `let` $x = (x : \sigma)$ `in` a. The derived typing rule is:

FUN*

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

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

The derived reduction rule is $(\lambda(x : \exists\,(Q)\ \sigma)\ a)\ v \xrightarrow{\beta_\star}$ `let` $x = (v : \exists(Q)\ \sigma)$ `in` a. Values must then be extended with expressions of the form $\lambda(x : \exists\,(Q)\ \sigma)\ a$, indeed.

5 Comparison with System-F

We have already seen that all ML programs can be written in ML^{F} without annotations. In Section 5.1, we provide a straightforward compositional translation of terms of System-F into ML^{F}. In Section 5.2, we then identify a subsystem of ML^{F}, called Shallow-$\mathrm{ML}^{\mathrm{F}}_{\star}$, whose let-binding free version is exactly the target of the encoding of System-F.

5.1 Encoding System-F into ML^{F}

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

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

The translation of types of System-F into ML^{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 types are translated to unconstrained bounds.

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

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

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

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

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

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

$$[\![\Lambda(\alpha) M]\!] = [\![M]\!] \qquad [\![M\ t]\!] = [\![M]\!] \qquad [\![x]\!] = x$$

$$[\![M\ M']\!] = [\![M]\!]\ [\![M']\!] \qquad [\![\lambda(x : t)\ M]\!] = \lambda(x : [\![t]\!])\ [\![M]\!]$$

Finally, we can state the following lemma:

LEMMA 8. *For any closed typing context 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 $[\![t]\!] \sqsubseteq \tau$.*

Remarkably, translated terms contain strictly fewer annotations than original terms—a property that was not true in Poly-ML. In particular, all type Λ-abstractions and type applications are dropped and only annotations of λ-bound variables remain. Moreover, some of these annotations are still superfluous.

5.2 Shallow-$\mathrm{ML}^{\mathrm{F}}_{\star}$

Types whose flexible bounds are always $\bot$ are called F-types (they are the translation of types of System F). Types of the form $\forall(\alpha \geq \sigma)\ \tau$, where σ 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, they are never used in the encoding of System-F. Are they needed as annotations?

Let a type be *shallow* if and only if all its rigid bounds are F-types. More generally, prefixes, typing contexts, and judgments are shallow if and only if they contain only shallow types. A derivation is shallow if all its judgments are shallow and Rule ORACLE is only applied to F-types. Notice that the explicit annotation (σ) has a shallow type if and only if σ is an F-type. We call Shallow-$\mathrm{ML}^{\mathrm{F}}_{\star}$ the set of terms that have shallow derivations. Interestingly, subject reduction also holds for Shallow-$\mathrm{ML}^{\mathrm{F}}_{\star}$.

Let-bindings do not increase expressiveness in $\mathrm{ML}^{\mathrm{F}}_{\star}$, since they can always be replaced by λ-bindings with oracles or explicit annotations. This is not true for Shallow-$\mathrm{ML}^{\mathrm{F}}_{\star}$, since shallow-types that are not F-types cannot be used as annotations. Therefore, we also consider the restriction Shallow-F of Shallow-$\mathrm{ML}^{\mathrm{F}}_{\star}$ to programs without let-bindings.

The encoding of System-F into ML^{F} given in Section 5.1 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.

6 Discussion

6.1 Expressiveness of ML^{F}

By construction, we have the chain of inclusions Shallow-F $\subseteq$ Shallow-ML$^{F}_{\star}$ $\subseteq$ ML$^{F}_{\star}$. We may wonder whether these languages have strictly increasing power. That is, ignoring annotations and the difference in notation between let-bindings and λ-redexes, do they still form a strict chain of inclusions? We conjecture that this is true.

Still, ML$^{F}_{\star}$ remains a second-order system and in that sense should not be *significantly* more expressive than System F. In particular, we conjecture that the term $(\lambda(y)\ y\,I\,;y\,K)\ (\lambda(x)\ x\,x)$ that is typable in F$^{\omega}$ but not in F [9] is not typable in ML^{F} either. Conversely, we do not know whether there exists a term of ML^{F} that is not typable in F$^{\omega}$.

Reducing all let-bindings in a term of Shallow-ML$^{F}_{\star}$ produces a term in Shallow-F. Hence, terms of Shallow-ML$^{F}_{\star}$ are strongly normalizable. We conjecture that so are all terms of 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 δ-rules. These could include primitive types (such as integers, strings, *etc.*) and operations over them. Sums and products, as well as predefined datatypes, can also be treated in this manner, but some extension is required to declare new data-types within the language itself.

The value restriction of polymorphism [28] that allows for safe mutable data-structures in ML should carry over to ML^{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^{F}, as it is in Poly-ML, which uses polymorphism extensively. An interesting relaxation of the value-only restriction has been recently proposed [6] and allows to always generalize type variables that never appears on the left hand-side of an arrow type; this gave quite satisfactory results in the context of Poly-ML and we can expect similar benefits for ML^{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.The closest of them is unquestionably Poly-ML [7], with which close connections have already been made. Poly-ML also subsumes previous proposals that encapsulate first-class polymorphic values within datatypes [25]. Odersky and Laufer's proposal [20] 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^{F}.

Rank-2 polymorphism actually allows for full type inference [13, 11]. However, the algorithm proceeds by reduction on source terms and is not very intuitive. Rank-2 polymorphism has also been incorporated in the Hugs implementation of Haskell [17], but with explicit type annotations. The GHC implementation of Haskell has recently been released with second-order polymorphism at arbitrary ranks [8]; 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 [14].

Although our proposal relies on the let-binding mechanism to introduce implicit polymorphismand flexible bounds in types to factorize all ways of obtaining type instances, there may still be some connection with intersection types [12], which we would like to explore. Our treatment of annotations as type-revealing primitives also resembles retyping functions (functions whose type-erasure η-reduces to the identity) [19]. However, our annotations are explicit and contain only certain forms of retyping functions. Type inference for System F modulo η-expansion is known to be undecidable as well [27].

Several people have considered partial type inference for System F [10, 1, 23] and stated undecidability results for some particular variants that in all cases amount—directly or indirectly—to permitting (and so forcing) 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 by Pfenning [22]. Despite our opposite choice, that is not to support second-order unification, there are at least two comparisons to be made. Firstly, Pfenning's work does not cover the language ML *per se*, but only the λ-calculus, since let-bindings are expanded prior to type inference. Indeed, ML is not the simply-typed λ-calculus and type inference in ML cannot, *in practice*, be reduced to type inference in the simply-typed λ-calculus after expansion of let-bindings. Secondly, one proposal seems to require annotations exactly where the other can skip them: in [22], markers (but no type) annotations must replace type-abstraction and type-application nodes; conversely, this information is omitted in ML^{F}, but instead, explicit type information must remain for (some) arguments of λ-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. We would like to understand our notion of unification 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 [5]. Flexible bounds might partly 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 [18]. However, our notion of prefix and its role in abstracting polytypes is quite different.

Actually, none of the above works did consider subtyping at all. This is a significant difference with proposals based on local type inference [2, 24, 21] 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^{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. Indeed, we implemented a prototype of ML^F and verified on a variety of examples that few annotations are actually required and always at predictable places. However, further investigations are still needed regarding the syntactic-value polymorphism restriction and its possible relaxation.

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 abstraction and instance relations.

7 References

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

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

[3] R. D. Cosmo. *Isomorphisms of Types: from lambda-calculus to information retrieval and language design*. Birkhauser, 1995.

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

[5] G. Dowek, T. Hardin, C. Kirchner, and F. 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.

[6] J. Garrigue. Relaxing the value-restriction. Presented at the third Asian workshop on Programmaming Languages and Systems (APLAS), 2002.

[7] J. Garrigue and D. 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.

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

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

[10] J. James William O'Toole and D. 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.

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

[12] T. 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.

[13] A. J. Kfoury and J. 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*, 1994.

[14] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 161–174. ACM, Jan. 1999.

[15] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, Sept. 1994.

[16] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. 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.

[17] 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.

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

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

[20] M. Odersky and K. Läufer. Putting type annotations to work. In *Proceedings of the 23rd ACM Conference on Principles of Programming Languages*, pages 54–67, Jan. 1996.

[21] M. Odersky, C. Zenger, and M. Zenger. Colored local type inference. *ACM SIGPLAN Notices*, 36(3):41–53, Mar. 2001.

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

[23] F. 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.

[24] B. C. Pierce and D. 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.

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

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

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

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

A Unification algorithm

The algorithm `unify` is specified in Section 1.5; it takes a prefix Q and two types τ and τ' and returns a prefix that unifies τ and τ' under Q (as described in Theorem 1) or fails. In fact, the algorithm `unify` is recursively defined by an auxiliary unification algorithm for polytypes: `polyunify` takes a prefix Q and two type schemes σ_1 and σ_2 and returns a pair (Q', σ') such that $Q \sqsubseteq Q'$ and $(Q')\ \sigma_1 \sqsubseteq \sigma'$ and $(Q')\ \sigma_2 \sqsubseteq \sigma'$.

The algorithms `unify` and `polyunify` are described in Figure 8. For the sake of comparison with ML, think of the input prefix Q

Figure 8. Unification 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 $Q$ in
    • let $Q^{i+1}$ be unify $(Q^i, \tau_1^i, \tau_2^i)$ for $1 \leqslant i \leqslant n$ in
    • 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 $\sigma \notin \mathcal{T}$
    • let $(Q', \_)$ be polyunify $(Q, \sigma, \tau)$ in
    • fail if $(\alpha \diamond \sigma) \notin Q'$.
    • return $(Q') \Leftarrow (\alpha = \tau)$
Case $(\alpha_1, \alpha_2)$ when $(\alpha_1 \diamond_1 \sigma_1) \in Q$ and $(\alpha_2 \diamond_2 \sigma_2) \in Q$ and
                                    $\alpha_1 \neq \alpha_2$ and $\sigma_1$, $\sigma_2$ are not in $\mathcal{T}$.
    • let $(Q', \sigma_3)$ be polyunify $(Q, \sigma_1, \sigma_2)$ in
    • fail if $(\alpha_1 \diamond_1 \sigma_1) \notin Q'$ or if $(\alpha_2 \diamond_2 \sigma_2) \notin Q'$
    • return $(Q') \Leftarrow (\alpha_1 \diamond_1 \sigma_3) \Leftarrow (\alpha_2 \diamond_2 \sigma_3) \Leftarrow \alpha_1 \wedge \alpha_2$.


polyunify $(Q, \sigma_1, \sigma_2)$
  — requires $\sigma_1$, $\sigma_2$, and all bounds in Q to be in normal form[1] :
Case $(\bot, \sigma)$ or $(\sigma, \bot)$: return $(Q, \sigma)$
Case $(\forall (Q_1)\ \tau_1, \forall (Q_2)\ \tau_2)$ with $Q_1$, $Q_2$, and $Q$ having disjoint
    domains (which usually requires renaming $\sigma_1$ and $\sigma_2$)
    • let $Q_0$ be unify $(QQ_1Q_2, \tau_1, \tau_2)$ in
    • let $(Q_3, Q')$ be $Q_0 \uparrow \mathsf{dom}(Q)$ in
    • return $(Q_3, \forall (Q')\ \tau_1)$
```

---

[1] Actually, only need to replace types of the form $\forall (\alpha \diamond \sigma)\ \alpha$ by σ, which can always be done lazily.

given to `unify` as a substitution and of the result prefix Q' as an instance of Q (*i.e.* a substitution of the form $Q'' \circ Q$) that unifies τ and τ'. First-order unification of polytypes essentially follows the general structure of first-order unification of monotypes. The main differences are that (i) the computation of the unifying substitution is replaced by the computation of a unifying prefix, (ii) additional work must be performed when a variable bound to a strict polytype (*i.e.* other than $\bot$ and not equivalent to a monotype) is being unified: bounds must be further unified (last case of `polyunify`) and the prefix updated accordingly. Auxiliary algorithms are used for this purpose.

Let a *rearrangement* of a prefix Q be a prefix equivalent to Q obtained by a permutation of bindings of Q.

DEFINITION 9. The *split* algorithm $Q \uparrow \bar{\alpha}$ takes a prefix Q and returns a pair of prefixes (Q_1, Q_2) such that (i) $Q_1 Q_2$ is a rearrangement of Q, (ii) $\bar{\alpha} \subseteq \mathsf{dom}(Q_1)$, and (iii) $\mathsf{dom}(Q_1)$ is minimal (in other words, Q_1 contains only bindings of Q useful for exporting the interface $\bar{\alpha}$, and Q_2 contains the rest). □

Figure 9. Algorithm W^{F}

```
infer $(Q, \Gamma, a)$:
  — proceeds by case analysis on expression a :
Case $x$: return $Q, \Gamma(x)$
Case $\lambda(x)\ a$:
    • let $Q_1 = (Q, \alpha \geq \bot)$ with $\alpha \notin \mathsf{dom}(Q)$
    • let $(Q_2, \sigma) = infer\ (Q_1, \Gamma, x : \alpha, a)$
    • let $\beta \notin \mathsf{dom}(Q_2)$ and $(Q_3, Q_4) = Q_2 \uparrow \mathsf{dom}(Q)$
    • return $Q_3, \forall (Q_4)\ \forall (\beta \geq \sigma)\ \alpha \to \beta$
Case $a\ b$:
    • let $(Q_1, \sigma_a) = infer\ (Q, \Gamma, a)$
    • let $(Q_2, \sigma_b) = infer\ (Q_1, \Gamma, b)$
    • let $\alpha_a, \alpha_b, \beta \notin \mathsf{dom}(Q_2)$
    • let $Q_3 =$ unify $((Q_2, \alpha_a \geq \sigma_a, \alpha_b \geq \sigma_b, \beta \geq \bot),$
                                                    $\alpha_a, \alpha_b \to \beta)$
    • let $(Q_4, Q_5) = Q_3 \uparrow \mathsf{dom}(Q)$
    • return $(Q_4, \forall (Q_5)\ \beta)$
Case let $x = a_1$ in $a_2$:
    • let $(Q_1, \sigma_1) = infer\ (Q, \Gamma, a_1)$
    • return $infer\ (Q_1, (\Gamma, x : \sigma_1), a_2)$
```

DEFINITION 10. The *abstraction-check* algorithm $(Q)\ \sigma \sqsubseteq^{?} \sigma'$ takes a prefix Q and two polytypes σ and σ' such that $(Q)\ \sigma \sqsubseteq \sigma'$ and checks that $(Q)\ \sigma \sqsubseteq \sigma'$ or fails otherwise. □

DEFINITION 11. The *update* algorithm $Q \Leftarrow (\alpha \diamond \sigma)$ takes a prefix Q and a binding $(\alpha \diamond \sigma)$ such that α is in the domain of Q and returns a prefix $(Q_0, \alpha \diamond \sigma, Q_1)$ such that (i) $(Q_0, \alpha \diamond' \sigma', Q_1)$ is a rearrangement of Q and (ii) $\mathsf{dom}(Q_1) \cap \mathsf{ftv}(\sigma) = \emptyset$. The algorithm fails when there is not such decomposition (because of circular dependencies) or when $\diamond'$ is $=$ and $(Q)\ \sigma' \sqsubseteq^{?} \sigma$ fails. □

DEFINITION 12. The *merge* algorithm $Q \Leftarrow \alpha \wedge \alpha'$ takes two variables α and α' and a prefix of the form $(Q_0, \alpha \diamond \sigma, Q_1, \alpha' \diamond' \sigma', Q_2)$ and returns the prefix $(Q_0, \alpha \diamond'' \sigma, Q_1, \alpha' = \alpha, Q_2)$ where $\diamond''$ is $\geq$ if both $\diamond$ and $\diamond'$ are $\geq$, and $\diamond''$ is $=$ otherwise. □

The implementation of algorithms *split*, *update*, and *merge* is straightforward. The algorithm *abstraction-check* can be reduced to a simple check on the structure of paths, thanks to the assumption $(Q)\ \sigma \sqsubseteq \sigma'$. By lack of space, they are all omitted.

B Type inference algorithm

Figure 9 defines the type-inference algorithm W^{F} for ML^{F}. The algorithm follows the algorithm W for ML, with only two differences: first, the algorithm builds a prefix Q instead of a substitution; second, all free type variables not in Γ are quantified at each abstraction or application. Since free variables of Γ are in $\mathsf{dom}(Q)$, finding quantified variables consists in splitting the current prefix according to $\mathsf{dom}(Q)$, as described by Definition 9.