Extending ML with Semi-Explicit Higher-Order Polymorphism

Jacques Garrigue[1] and Didier Rémy[2]

[1] Kyoto University Research Institute for Mathematical Sciences, Kitashirakawa-Oiwakecho, Sakyo-ku, Kyoto 606-01, Japan
[2] INRIA-Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France

Abstract. We propose a modest conservative extension to ML that allows semi-explicit higher-order polymorphism while preserving the essential properties of ML. In our proposal, the introduction of polymorphic types remains fully explicit, that is, both the introduction and the exact polymorphic type must be specified. However, the elimination of polymorphic types is now semi-implicit: only the elimination itself must be specified as the polymorphic type is inferred. This extension is particularly useful in Objective ML where polymorphism replaces subtyping.

Introduction

The success of the ML language is due to its combination of several attractive features. Undoubtedly, the polymorphism of ML [2] —or *polymorphism* à la *ML*— with the type inference it allows, is a major advantage. The ML type system stays in close correspondence with the rules of logic, following the Curry-Howard isomorphism between types and formulas, which provides a simple intuition, and a strong type discipline. Simultaneously, type inference relieves the user from the burden of writing types: an algorithm automatically checks whether the program is well-typed and, if true, it returns a principal type.

Upon this simple system, many extensions have been proposed: polymorphic records, first-class continuations, first-class abstract datatypes, type-classes, overloading, objects, etc. In all these extensions, type inference remains straightforward first-order unification with toplevel polymorphism. This shows the robustness of ML-style type inference.

There are of course cases where one would like to have higher-order polymorphism, as in system F. ML allows for polymorphic definitions, but abstractions can only be monomorphic. Traditionally, ML polymorphism is used for definitions of higher-order functions such as folding or iteration over a parameterized datatype. Some higher-order functionals require polymorphic functions as arguments. These situations mostly appear in encodings, and cases that appear in real programs can usually be solved by using functors of the module language.

This simple picture, which relies on a clear separation between data and functions operating on data, has recently been invalidated by several extensions. For instance, data and methods are packed together inside objects. This decreases the need for polymorphism, since methods can be specialized to the piece of data they are embedded with. However, data transformers such as folding functions remain parametric in the type of the output. For instance, a function `fold` with the ML type $\forall \beta, \alpha.\ \beta\ \texttt{list} \rightarrow (\beta \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ should become a method for container objects, of type $\forall \alpha.\ (\tau \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$ where τ is the type of the elements of the container. The extension of ML with first-class abstract types [6, 14] also requires first-class polymorphic functions: for instance, an expression such as $\lambda f.\, \texttt{open}\ x\ \texttt{as}\ y\ \texttt{in}\ f\ y$ can only be typed if the argument f is polymorphic so that the abstract representation of y is not revealed outside the scope of the open construct.

First-class polymorphic values have been proposed in [14, 9] following the ideas developed by Laüfer & Odersky in [6]. After de-sugaring, these proposals all reduce to the same idea of using explicit, mutually inverse introduction and elimination functions to coerce higher-order types into basic, parameterized type symbols and back. Therefore, they all suffer the same problem: types must be written explicitly, both at the introduction and elimination of polymorphism.

Recent results on the undecidability of type inference for system F [16, 5, 11] also do not leave many hopes for finding a good subset of system F that significantly extends ML, moreover with decidable type inference and principal types. Previous attempts to accomplish this task were unsuccessful.

This is not the path we choose here. On the contrary, we do not infer higher-order types and avoid higher-order unification, undecidable in general. We also maintain the simplicity of the ML type system, following the premise that an extension of ML should not modify the ML polymorphism in its essence, even an extension that actually increases the level of polymorphism.

The original insight of our work is that, although ML polymorphism allows type inference, actual ML programs do already contain a lot of type information. All constants, all constructors, and all previously defined functions have already known types. This information only longs to be used appropriately.

In comparison to previous works, we remove the need for type annotations at the elimination of polymorphism, using type inference to propagate explicit type information between different points of the program. In our proposal, it becomes unnecessary to tag values of polymorphic types with type symbols. A type annotation at the introduction of a polymorphic value is sufficient and can be propagated to the elimination site (following the data-flow view of programs). This makes the handling of such values considerably easier, and reasonably practical for use in a programming language.

In a first section, we present our solution informally, and explain how it simplifies the use of higher-order types in ML. Then, we develop this approach formally, proving all fundamental properties. In a third section, encodings are provided, both for previous formulations of first-class polymorphism, and for system F itself. In the next section, we present two extensions to the core language. The first extension restricts polymorphism to values. The second one allows a more comfortable syntax to manipulate objects with polymorphic methods.

Lastly, we compare with related works, and conclude. Proofs of main theorems are given in appendix.

1 Informal approach

In this section we will present our solution informally. We first present a naive straightforward proposal. We show that this solution needs to be restricted to avoid higher-order unification. Last, we describe a simple solution that allows for complete type inference.

1.1 A naive solution

Naively, ML types can be easily extended with polymorphic types. A typical program that cannot be typed in ML and could be typed in system F is $\lambda f.\, f f$. This expression is not very interesting for itself. However, a few variations are sufficient to illustrate most aspects of type inference in the presence of higher-order types. Useful examples can be found in section 4.2 in addition to those suggested in the introduction.

Although $\lambda f.\, f f$ is not typable in ML, the expression $\texttt{let}\ f = \lambda x.\, x\ \texttt{in}\ f\ f$ is. One can see let-definitions as a special syntax, combined with a special typing rule, for the application $(\lambda f.\, f f)\ (\lambda x.\, x)$. Let us exercise by replacing the Let polymorphic binding by first-class polymorphism. The identity $\lambda x.\, x$ of type $\alpha \to \alpha$ has also type scheme $\forall \alpha . \alpha \to \alpha$. We shall write $[\lambda x.\, x : \forall \alpha . \alpha \to \alpha]$ for the *creation* (or *introduction*) of the polymorphic value $\lambda x.\, x$ with type scheme $\forall \alpha . \alpha \to \alpha$. In order to avoid confusion with ML types, we explicitly coerce $\forall \alpha . \alpha \to \alpha$ to a regular ML type $[\forall \alpha . \alpha \to \alpha]$, adding the type constructor $[\_]$. We call $\forall \alpha . \alpha \to \alpha$ a *polymorphic type* or a *type scheme* and $[\forall \alpha . \alpha \to \alpha]$ a *polytype*.

Let f be the expression $[\lambda x.\, x : \forall \alpha . \alpha \to \alpha]$, which has type $[\forall \alpha . \alpha \to \alpha]$. As any first-class value, f can be passed to other functions, be stored in data-structures, etc. For instance $(f, 1)$ is a pair of type $([\forall \alpha . \alpha \to \alpha] \times \texttt{int})$. A polymorphic function (*e.g.* a polymorphic value that is a function) cannot be applied directly, since it is typed with a polytype, which is incompatible with an arrow type. We must previously *open* (or *eliminate*) the polytype. We introduce a new construct $\langle \_ \rangle$ for that purpose. Hence, $\langle f \rangle$ is a function of type an instance of the polymorphic type $\forall \alpha . \alpha \to \alpha$, *i.e.* $\tau \to \tau$ for some type τ. Its principal type is $\alpha \to \alpha$.

The raw expression $\lambda f.\, f f$ is not well typed. It should be passed a polymorphic value as argument, for instance, of type $[\forall \alpha . \alpha \to \alpha]$. Here, we shall introduce polymorphism only by a type constraint on the argument: $\lambda f{:}\, \forall \alpha . \alpha \to \alpha .\, \langle f \rangle\ f$. The first occurrence of f in the body is opened to eliminate polymorphism before it is applied. The following definition of g is well-typed

$$g \stackrel{\triangle}{=} \lambda f{:}\, [\forall \alpha . \alpha \to \alpha] .\, \langle f \rangle\ f : [\forall \alpha . \alpha \to \alpha] \to [\forall \alpha . \alpha \to \alpha]$$

So are the two following variants:

$$h \stackrel{\triangle}{=} \lambda f\colon [\forall \alpha.\alpha \to \alpha].\, \langle f \rangle\, \langle f \rangle : [\forall \alpha.\alpha \to \alpha] \to \alpha' \to \alpha'$$
$$k \stackrel{\triangle}{=} \lambda f\colon [\forall \alpha.\alpha \to \alpha].\, [\langle f \rangle\, \langle f \rangle : \forall \alpha.\alpha \to \alpha] : [\forall \alpha.\alpha \to \alpha] \to [\forall \alpha.\alpha \to \alpha]$$

In h, the occurrence of f in the argument position is also opened, so the result type is no longer a polytype. In k, polymorphism is lost as in h, then it is recovered explicitly. Finally, we can apply g to f:

$$(\lambda f\colon [\forall \alpha.\alpha \to \alpha].\, \langle f \rangle\ f)\ [\lambda x.\, x : \forall \alpha.\alpha \to \alpha] : [\forall \alpha.\alpha \to \alpha]$$

More interestingly, the following expression is also well-typed

$$(\lambda g.\, g\ [\lambda x.\, x : \forall \alpha.\alpha \to \alpha])\ (\lambda f\colon [\forall \alpha.\alpha \to \alpha].\, \langle f \rangle\ f)$$

There is no term typable in ML that has the same erasure (untyped λ-term) as the above term. Note that no type annotation is needed on g since although g has a polytype as result, it is never opened.

1.2 An obvious problem

The examples above mixed type-inference and type-checking (using type-annotations). The obvious problem of type inference in the presence of higher-order types remains to be solved: what happens when expressions of unknown type are opened. Should the program $\lambda f.\, \langle f \rangle\ f$ or simpler $\lambda x.\, \langle x \rangle$ be typed?

The answer is clearly negative, since this would amount to inferring higher-order types, which we choose to avoid here. We should keep all user-provided polymorphism, but never guess polymorphism.

The attempt to forbid lambda abstraction of unspecified type to be a polytype does not work. It would violate the assumption that polytypes are regular ML types. Thus, if $\lambda x.\, x$ has type $\alpha \to \alpha$, it should also have type $[\sigma] \to [\sigma]$ for any polymorphic type σ. Actually, it is important that $\lambda x.\, x$ has all these types. For instance, both $(\lambda x.\, x)\ f$ and $\lambda x.\, f\ x$ should be typable and have the same type as f.

When typing $\lambda f.\, \langle f \rangle\ f$, variable f is first given an unknown type α'. Guessing $\forall \alpha.\alpha \to \alpha$ for α' would be correct, but not principal. More subtle, the expression $\lambda f.\, \langle f \rangle\ (g\ f)$ may only be typed with $[\forall \alpha.\alpha \to \alpha] \to [\forall \alpha.\alpha \to \alpha]$ and has a principal derivation. However, we should also reject this program. Informally, type inference would imply backtracking: f is first assumed of unknown type α'; we cannot type $\langle f \rangle$ so we backtrack; typing the application $g\ f$ forces f to be of type $[\forall \alpha.\alpha \to \alpha]$, then $\langle f \rangle$ can be typed, and so on. This causes two problems. Firstly, backtracking may lead to a combinatorial explosion of the search space, and we would rather fail in every case where some inference order would fail. More theoretically, constraints imported from other branches may disappear whenever a reduction occurs on the expression containing them, making the term untypable by lack of principality. We would loose the subject reduction property.

1.3 A simple solution

The essence of our proposal is a simple mechanism based on unification that distinguishes polytypes that have been user-provided from those that have just been guessed. Each occurrence of a polytype $[\sigma]$ is labeled with a label ϵ. That is, we write $[\sigma]^\epsilon$ rather than $[\sigma]$. Actually, we keep $[\sigma]$ as an abbreviation for $[\sigma]^\epsilon$ where ϵ is an anonymous label, *i.e.* one that does not appear anywhere else. Intuitively, labels indicate sharing of polytype nodes.

The elimination of polymorphism $\langle a\rangle$ is possible whenever a can be typed with $[\sigma]^\epsilon$ where ϵ does not appear anywhere else. Informally, we could just say when a has polytype $[\sigma]$ (since ϵ is anonymous). The intuition is that an anonymous label ϵ ensures that the corresponding polytype does not appear anywhere else and *a fortiori* does not appear as an hypothesis (*i.e.* in a negative occurrence, such as the context or the left hand-side of an arrow); thus, it must have been user-provided.

For instance, in the expression $\lambda f.\, \langle f\rangle\ f$, the lambda-bound variable f can be given the polytype $[\forall\alpha.\alpha \to \alpha]^\epsilon$, with a monomorphic ϵ; since all instances of f will share the same label ϵ, the label cannot be anonymous as required when typing $\langle f\rangle$. Indeed, the type of the variable f in $\langle f\rangle$ is a polytype only under the assumption that the binding occurrence of f is typed with exactly the same polytype.

On the contrary, when a polytype has been confirmed, we want to propagate it, following the definition order. We use polymorphism to generate new anonymous labels from older ones. We allow quantification on anonymous labels, and later instantiation of quantified labels to new anonymous labels.

When typing the expression $\lambda f\!:[\forall\alpha.\alpha \to \alpha].\, \langle f\rangle\ f$, the type assumption $f : \forall\epsilon.[\forall\alpha.\alpha \to \alpha]^\epsilon$ is added to the context in which $\langle f\rangle\ f$ is typed. Thus, variable f has type $[\forall\alpha.\alpha \to \alpha]^{\epsilon_1}$ with a different, anonymous, label ϵ_1, and therefore $\langle f\rangle$ is well-typed. For technical reasons we chose not to allow type annotation of abstracted variables in our system, but instead $\lambda x\!:\tau.\, a$ can be seen as $\lambda x.$ `let` $x = (x : \tau)$ `in` a. Type annotation $(\_ : \tau)$ renames all ϵ's free in τ into fresh ones.

2 Formal approach

We formalize our approach as a small extension to core ML.

2.1 The core language

Types We assume given two collections of type variables $\alpha \in \mathcal{V}$, and labels $\epsilon \in \mathcal{E}$. The syntax of types is:

$\tau ::= \alpha \mid \tau \to \tau \mid [\sigma]^\epsilon$	Monotypes
$\sigma ::= \tau \mid \forall\alpha.\sigma$	Type schemes
$\varsigma ::= \sigma \mid \forall\epsilon.\varsigma$	Generic schemes
$\xi ::= \alpha \mid \epsilon$	Variables

The construct $[\sigma]^\epsilon$ is used to coerce a type scheme σ to a monotype. We call $[\sigma]^\epsilon$ a weak polytype. The label ϵ is used to keep track of sharing between weak polytypes, or allow them to be usable polytypes, when it is quantified as $\forall\epsilon.[\sigma]^\epsilon$. We do not quantify labels in σ, since this would not add any power to the system (it would be redundant with explicit type annotations).

Free type variables and free labels of a generic scheme, type scheme, or monotype ς are written $FV(\varsigma)$ and $FL(\varsigma)$, and are defined as usual. In a type scheme $\forall\xi.\varsigma$, $\forall$ acts as a quantifier, and the variable or label ξ is bound (*i.e.* not free) in $\forall\xi.\varsigma$. We consider type schemes equal by renaming of bound variables and labels. As usual substitutions leave bound variables and labels unchanged. For example $(\alpha \to [\forall\alpha'.\alpha' \to \alpha]^\epsilon)\{\tau/\alpha\}$ is $\tau \to [\forall\alpha'.\alpha' \to \tau]^\epsilon$ provided α' is not free in τ. An instance of a type scheme $\forall\bar\epsilon,\bar\alpha.\tau_0$ is $\tau\{\bar\epsilon',\bar\tau/\bar\epsilon,\bar\alpha\}$.

Expressions

$$\begin{aligned} a ::= {} & x \mid \lambda x.\, a \mid a\ a \mid \texttt{let}\ x = a\ \texttt{in}\ a \\ & \mid [a : \sigma] \mid \langle a \rangle \mid (a : \tau) \end{aligned}$$

The first line corresponds exactly to core ML. We then introduce three new constructs: introduction and elimination of first-class polymorphism and type annotation.

Typing rules are given in figure 1. All typing rules but the last three ones are

$$\text{(Var)}\ \frac{x : \varsigma \in A}{A \vdash x : \varsigma} \qquad \text{(Fun)}\ \frac{A \oplus x : \tau_0 \vdash a : \tau}{A \vdash \lambda x.\, a : \tau_0 \to \tau} \qquad \text{(App)}\ \frac{A \vdash a_1 : \tau_2 \to \tau_1 \qquad A \vdash a_2 : \tau_2}{A \vdash a_1\ a_2 : \tau_1}$$

$$\text{(Gen-V)}\ \frac{A \vdash a : \sigma \qquad \alpha \notin FV(A)}{A \vdash a : \forall\alpha.\sigma} \qquad \text{(Gen-E)}\ \frac{A \vdash a : \varsigma \qquad \epsilon \notin FL(A)}{A \vdash a : \forall\epsilon.\varsigma} \qquad \text{(Inst-V)}\ \frac{A \vdash a : \forall\alpha.\sigma}{A \vdash a : \sigma\{\tau/\alpha\}}$$

$$\text{(Inst-E)}\ \frac{A \vdash a : \forall\epsilon.\varsigma}{A \vdash a : \varsigma\{\epsilon'/\epsilon\}} \qquad \text{(Let)}\ \frac{A \vdash a_1 : \varsigma \qquad A \oplus x : \varsigma \vdash a_2 : \tau}{A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau}$$

$$\text{(Ann)}\ \frac{A \vdash a : \tau_1 \qquad (\tau_1 : \tau : \tau_2)}{A \vdash (a : \tau) : \tau_2} \qquad \text{(Poly)}\ \frac{A \vdash a : \sigma_1 \qquad (\sigma_1 : \sigma : \sigma_2)}{A \vdash [a : \sigma] : [\sigma_2]^\epsilon}$$

$$\text{(Use)}\ \frac{A \vdash a : \forall\epsilon.[\sigma]^\epsilon}{A \vdash \langle a \rangle : \sigma}$$

Fig. 1. Typing rules

quite standard. Rules Ann and Poly use an auxiliary relation $(\_ : \_ : \_)$. Given a type scheme σ, we write $(\sigma_1 : \sigma : \sigma_2)$ if there exists a substitution θ from type variables to types and two substitutions ρ_1 and ρ_2 from labels to labels, such that $\sigma_1 = \theta(\rho_1(\sigma))$ and $\sigma_2 = \theta(\rho_2(\sigma))$. The intuition is that if θ is the identity, then σ_1 and σ_2 are both equal to σ except maybe in their labels. Indeed, $(\rho_1(\sigma) : \sigma : \rho_2(\sigma))$. If σ does not contain any label, then $(\sigma_1 : \sigma : \sigma_2)$ is equivalent to σ_1 and σ_2 being the same generic instance of σ. An important property of the relation $(\_ : \sigma : \_)$ is its stability by substitution. That is, if $(\sigma_1 : \sigma : \sigma_2)$, then $(\theta(\sigma_1) : \sigma : \theta(\sigma_2))$ for any substitution θ.

This relation is used to type explicit annotations. The construct $(\_ : \tau)$ could have been replaced by a countable collection of primitives $\lambda x.\, (x : \tau)$ indexed by τ and given with principal type schemes $\forall \bar\epsilon_1, \bar\epsilon_2, FV(\tau).\ \tau\{\bar\epsilon_1/\bar\epsilon\} \to \tau\{\bar\epsilon_2/\bar\epsilon\}$ where $\bar\epsilon_1$ and $\bar\epsilon_2$ are different renamings of the labels $\bar\epsilon$. That is, to type an expression $(a : \tau)$, let τ_1 and τ_2 be two copies of τ where labels have been been renamed, and θ be a substitution such that a has type $\theta(\tau_1)$; then $(a : \tau)$ has type $\theta(\tau_2)$.

Rule Poly uses the same relation, except that types schemes replace types. To type $[a : \sigma]$, let σ_1 and σ_2 be two copies of σ where labels have been been renamed; find a generic instance σ' of the principal type scheme of a and a substitution θ such that $\theta(\sigma')$ and $\theta(\sigma_1)$ are equal, and return $[\theta(\sigma_2)]^\epsilon$.

Last, rule Use says that polymorphism can be used only if the label of the polytype does not occur anywhere else.

As an example, we have the following derivation, where σ abbreviates $\forall\alpha.\alpha \to \alpha$ and A is $f : [\sigma]^{\epsilon_1}$:

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{
          (\textsc{Var})\ \dfrac{}{A \vdash f : [\sigma]^{\epsilon_1}} \qquad ([\sigma]^{\epsilon_1} : [\sigma]^{\epsilon} : [\sigma]^{\epsilon_2})
        }{A \vdash (f : [\sigma]^\epsilon) : [\sigma]^{\epsilon_2}}\ (\textsc{Ann})
      }{A \vdash (f : [\sigma]^\epsilon) : \forall \epsilon_2 . [\sigma]^{\epsilon_2}}\ (\textsc{Gen-E})
    }{A \vdash \langle f : [\sigma]^\epsilon \rangle : \forall\alpha.\alpha \to \alpha}\ (\textsc{Use})
  }{A \vdash \langle f : [\sigma]^\epsilon \rangle : [\sigma]^{\epsilon_1} \to [\sigma]^{\epsilon_1}}\ (\textsc{Inst-V})
}{}
$$

$$
\dfrac{
  \dfrac{
    \vdots \qquad \dfrac{}{A \vdash f : [\sigma]^{\epsilon_1}}\ (\textsc{Var})
  }{A \vdash \langle f : [\sigma]^\epsilon \rangle\ f : [\sigma]^{\epsilon_1}}\ (\textsc{App})
}{\vdash \lambda f.\, \langle f : [\sigma]^\epsilon \rangle\ f : [\sigma]^{\epsilon_1} \to [\sigma]^{\epsilon_1}}\ (\textsc{Fun})
$$

2.2 Dynamic semantics

We give a call-by-value semantics for the core language. Values and evaluations contexts are:

$$
\begin{aligned}
v &::= w \mid [v : \sigma] \\
w &::= \lambda x.\, a \mid (w : \tau_1 \to \tau_2) \\
E &::= \{\} \mid E\ a \mid v\ E \mid \texttt{let}\ x = E\ \texttt{in}\ a \mid [E : \sigma] \mid (E : \tau) \mid \langle E \rangle
\end{aligned}
$$

One step is either a reduction of the form:

$$\begin{aligned}
(\lambda x.\, a)\ v &\xrightarrow{\mathsf{Fun}} a\{v/x\} \\
\mathtt{let}\ x = v\ \mathtt{in}\ a &\xrightarrow{\mathsf{Let}} a\{v/x\} \\
\langle [v : \forall \bar{\alpha}.\tau] \rangle &\xrightarrow{\mathsf{Use}} (v : \tau) \\
(v_1 : \tau_2 \to \tau_1)\ v_2 &\xrightarrow{\mathsf{Tfun}} (v_1\ (v_2 : \tau_2) : \tau_1) \\
([v : \forall \bar{\xi}.\tau] : [\sigma]^{\epsilon}) &\xrightarrow{\mathsf{Tpol}} [(v : \tau) : \sigma] \\
(v : \alpha) &\xrightarrow{\mathsf{Tvar}} v
\end{aligned}$$

or an inner reduction obtained by induction:

$$\frac{a_1 \xrightarrow{\mathsf{r}} a_2}{E\{a_1\} \xrightarrow{\mathsf{r}} E\{a_2\}}$$

Note that α, in rule TVAR, is really a variable and not a meta-variable. It is a major difference with ML that type annotations are not just a means to restrict principal types to instances. On the opposite, they allow better typings. Thus, reduction must preserve type annotations as long as they provide useful typing information. Indeed, while terms are only reduced by rules FUN, LET, and USE, we need the rules TFUN and TPOL to maintain this type information. Rule TVAR erases empty type information. Although types are preserved during reduction, they do not actually participate in the reduction. In particular, it would be immediate to define an untyped reduction $\Longrightarrow$ and a type-erasure $\mathcal{E}$, and to show that if $a_1 \longrightarrow a_2$, then $(\mathcal{E}(a_1) \Longrightarrow \mathcal{E}(a_2))$ or $\mathcal{E}(a_1)$ and $\mathcal{E}(a_2)$ are equal.

2.3 Type soundness

We could easily show that evaluation cannot go wrong by means of translation in system F. Subject reduction is an intermediate result of a direct proof that is neither required nor implied by type soundness. However, it is quite important for itself, since it shows that each reduction step preserves typings, and thus that the static semantics is tightly related to the dynamic semantics.

Subject reduction is not obviously preserved by extension to polytypes: the new constructions allow more programs to be typed, but simultaneously the reduction of those expressions requires more programs to be typable. In particular, subject reduction would not hold if we threw away type constraints too early during reduction.

Both subject reduction and type inference are simplified by restricting ourselves to canonical derivations. A similar result existed for the original Damas-Milner presentation of ML, but ML is now often presented in its syntax directed form.

Canonical derivations are those where occurrences of rules GEN and INST are restricted as follows:

– rule GEN only occurs as the last rule of the derivation or right above rule POLY, USE, the left premise of rule LET, or another rule GEN.

– rule Inst may only occur right after rule Tvar, rule Use, or another rule Inst.

Lemma 1 (Canonical derivations). *A valid typing judgment $A \vdash a : \tau$ has a canonical derivation.*

Another classical result is the stability of typing judgments by substitution:

Lemma 2 (Stability). *If $A \vdash a : \tau$, then for any substitution θ, $\theta(A) \vdash a : \theta(\tau)$.*

It is important to notice that the substitution is not applied to the expression a, in particular type constraints inside a are left unchanged: their free variables must be understood as if they were closed by existential quantification.

We define a relation $a_1 \subset a_2$ between programs stating that all typings of a_1 are also typings of a_2,*i.e.*

$$a_1 \subset a_2 \overset{\triangle}{=} (\forall A, \varsigma,\ A \vdash a_1 : \varsigma \Longrightarrow A \vdash a_2 : \varsigma)$$

Theorem 1 (Subject reduction). *Reduction preserves typings,* i.e. *if $a_1 \longrightarrow a_2$, then $a_1 \subset a_2$.*

Subject reduction is not sufficient to prove type soundness, since the full relation (every program has every type in any context) satisfies subject reduction but does not preserve from type errors. It must be complemented by the following result:

Theorem 2 (Canonical forms). *Irreducible programs that are well-typed in the empty environment are values.*

Type soundness is a straightforward combination of the two previous theorems.

2.4 Type inference

First-order unification on simple types must be extended to handle polytypes. During unification, a polytype is treated as a rigid skeleton corresponding to the polymorphic part, on which hang simple types. We present both unification and type inference as solving unification constraints following [4].

Unification on simple types First, we remind unification for simple types. In this part we exclude polytypes from types τ. A unification problem is a formula U defined by the following grammar.

$$\begin{array}{rll} U ::= \bot \mid \top \mid U \wedge U \mid \exists\, \alpha. U \mid e & \quad & \text{Unification problems} \\ e ::= \tau \mid \tau \doteq e & & \text{Multi-equations} \end{array}$$

The symbols $\top$ and $\bot$ are respectively the unsatisfiable and trivial unification problems. We treat them as a unit and a zero for $\wedge$. That is $U \wedge \top$ and $U \wedge \bot$ are respective equal to U and $\bot$. We also identify $\top$ with singleton multi-equations. That is, we can always consider that unification problems U contain at least one multi-equation $\alpha \doteq e$ for each variable of U. A complex formula is the conjunction of other formulas or the existential quantification of another formula. The symbol $\wedge$ is commutative and associative. The symbol $\exists$ acts as a binder, *i.e.* free variables of $\exists\,\alpha. U$ are free variables of U except α. Bound variables can freely be renamed. We identify $\exists\,\alpha_1.\exists\,\alpha_2.U$ and $\exists\,\alpha_2.\exists\,\alpha_1.U$ and simply write $\exists\,\alpha_1, \alpha_2.U$. The symbol $\doteq$ is associative and commutative. Thats is, multi-equations are in fact multi-sets of terms.

A substitution θ is a solution of a multi-equation if it sends all terms of the multi-equation to the same image. The substitution θ satisfies a conjunction of subproblems if it satisfies all subproblems; θ is a solution of $\exists\,\alpha.U$ if it can be extended on α into a solution of U.

Two unification problems are equivalent if they have the same set of solutions. One can check that all previous structural equalities are indeed equivalences.

Given a unification problem U, we define the containment ordering $\prec_U$ as the transitive closure of the immediate precedence ordering containing all pairs $\alpha \prec \alpha'$ such that there exists a multi-equation $\alpha \doteq \tau \doteq e$ in U where τ is a non-variable term that contains α'. A unification problem is strict if $\prec_U$ is strict. Remark that strictness is syntactic and is not preserved by equivalence. The idea is that strictness detects cycles, only on fully merged and decomposed unification problems.

A problem is in solved form if it is either $\perp$ or $\top$, or if it is strict and of the form $\exists\,\bar{\alpha}.\ \bigwedge_{i\in 1..n} e_i$, merged and decomposed. In particular, multi-equations e_i contain at most one non-variable term, and if $i \neq j$ then e_i and e_j contain no variable term in common. An explicit principal solution θ can be read straightforwardly from a problem in solved form. We write $U \Rrightarrow \exists\,\bar{\xi}.\theta$ if θ is a principal solution of U and variables $\bar{\xi}$ are not free in U, or by abuse of notation, if U is unsatisfiable and θ is $\perp$.

Occur-Check
    if $\prec_U$ is not strict **then**
    $U \Rrightarrow \perp$

Merge
    $\alpha \doteq e \wedge \alpha \doteq e' \Rrightarrow \alpha \doteq e \doteq e'$

Absorb
    $\alpha \doteq \alpha \doteq e \Rrightarrow \alpha \doteq e$

Decompose
    $\tau_1 \to \tau_2 \doteq \tau_1' \to \tau_2' \doteq e \Rrightarrow \tau_1 \to \tau_2 \doteq e \wedge \tau_1 \doteq \tau_1' \wedge \tau_2 \doteq \tau_2'$

Fig. 2. First-order unification of simple types

The unification algorithm is given as a set of rewriting rules that preserve equivalence in figure 2. There are implicit context rules that allow to rewrite complex formulas by rewriting any sub-formula. It is well-known that given an arbitrary unification problem, applying these rules always terminate with a unification problem in solved-formed. The rule Occur-Check rejects solutions with recursive types. If it were omitted the algorithm would infer recursive types.

Unification with polytypes We now allow polytypes $[\sigma]^\epsilon$. The rule Decompose is naturally extended by adding a case for polytypes. Thus we have to extend typing problems with equations between type schemes.

$$U ::= \ \dots \mid \sigma \doteq \sigma$$

Notice that these are not multi-equations, since a variable cannot be equated to a polymorphic type scheme, and as a result equations involving type schemes are never merged.

Decompose

$[\sigma]^\epsilon \doteq [\sigma']^{\epsilon'} \doteq e \Rrightarrow [\sigma]^\epsilon \doteq e \wedge \epsilon \doteq \epsilon' \wedge \sigma \doteq \sigma'$

Clash

$[\sigma]^\epsilon \doteq \tau \rightarrow \tau' \doteq e' \Rrightarrow \bot$

Polytypes

if $\bar\alpha \cap \bar\alpha' = \emptyset$ **then**

$\forall\bar\alpha.\tau \doteq \forall\bar\alpha'.\tau' \Rrightarrow \exists\,\bar\alpha,\bar\alpha'.\,\tau \doteq \tau' \wedge \bar\alpha \leftrightarrow \bar\alpha'$

Renaming-True

if $\bar\alpha = (\alpha_i)^{i\in 1..n+p}$ **and** $\bar\alpha' = (\alpha'_i)^{i\in 1..n+q}$ **then**

$\exists\,\bar\alpha,\bar\alpha'.\,(\alpha_i \doteq \alpha'_i)^{i\in 1..n} \wedge \bar\alpha \leftrightarrow \bar\alpha' \Rrightarrow \top$

Renaming-False

if $\beta \in \bar\alpha$ **and** $\tau \notin \bar\alpha' \cup \{\beta\}$ **then** $\beta \doteq \tau \doteq e \wedge \bar\alpha \leftrightarrow \bar\alpha' \Rrightarrow \bot$

if $\beta \in \bar\alpha \cap FV(\tau)$ **and** $\tau \neq \beta$ **then** $\beta' \doteq \tau \doteq e \wedge \bar\alpha \leftrightarrow \bar\alpha' \Rrightarrow \bot$

Fig. 3. First-order unification of simple types with polytypes

A substitution θ is solution of a polytype equation $\sigma \doteq \sigma'$ if $\theta(\sigma) = \theta(\sigma')$, where equality is the usual equality for type schemes in ML, *i.e.* it is taken modulo reordering and renaming of universal variables, and removal of useless universal variables. Clearly, $\forall\bar\alpha.\tau = \forall\bar\alpha'.\tau'$ if and only if there exists a substitution θ such that

1. $\theta(\tau) = \theta(\tau')$,
2. $\theta \upharpoonright \bar\alpha$ and $\theta \upharpoonright \bar\alpha'$ are injective in $\bar\alpha \cup \bar\alpha'$, and
3. no variable of $\bar\alpha \cup \bar\alpha'$ appears in $\theta \setminus (\bar\alpha \cup \bar\alpha')$.

We could have solved such unification problems by first unifying τ and τ' and then checking the constraints. However, this would force some unnecessary dependence. Indeed, the condition 1 above can be treated as a unification problem θ. We introduce another kind of unificands $\bar\alpha \leftrightarrow \bar\alpha'$ whose solutions are substitutions satisfying the conditions 2 and 3. We consider $\bar\alpha$ and $\bar\alpha'$ as multisets (*i.e.* the comma is associative and commutative). In order to avoid special cases, we require that no variable is listed twice in the sequence $\bar{\alpha}, \bar{\alpha}'$ (in particular $\bar{\alpha} \cap \bar{\alpha}'$ is empty). The symbols $\doteq$ (in polytype equations) and $\leftrightarrow$ are commutative.

Rules for unification with polytypes are those of figure 2 plus those of figure 3. Rule Clash handles type incompatibilities. Rule Polytypes transforms polytype equations as described above. Rule Renaming-True allows to remove a satisfiable renaming constraints that became independent. On the opposite, rule Renaming-False detects unsolvable renaming constraints. In the first case, a solution θ of $\bar{\alpha} \leftrightarrow \bar{\alpha}'$ would identify a variable β of $\bar{\alpha}$ with another variable of $\bar{\alpha}$ (thus θ would not be injective) or with a term outside of $\bar{\alpha} \cup \bar{\alpha}'$. In the second case, the image of a variable β' would contain properly a variable β of $\bar{\alpha}$, making it leak into a wider environment (thus, violating condition 3).

It can be easily checked that if U is merged and decomposed, then for every renaming constraint that remains either rule Renaming-True or -False applies. Therefore, renaming constraints can always be eliminated.

Type inference For type inference, we extend atomic formulas with typing problems. A typing problem is a triple, written $A \triangleright a : \tau$, of an environment A, a term a, and a type τ. A solution of a typing problem $A \triangleright a : \tau$ is a substitution θ such that $\theta(A) \vdash a : \theta(\tau)$. By lemma 2, the set of solutions of a typing problem is stable under substitutions. Thus, typing problems can be treated as unification problems, following [13]. The rules for solving typing problems are given in figure 4. The generalization *Gen* (σ, A) is, as usual, $\forall \bar{\xi}.\sigma$ where $\bar{\xi}$ are all free variables and free labels of σ that do not occur in A.

Theorem 3. *Given a typing problem* $(A \triangleright a : \tau)$ *there exists a principal solution, which is computed by the set of rules described in figures 2, 3 and 4.*

2.5 Printing labels as sharing constraints

We propose here an alternative interface to the system, potentially enhancing readability of types shown to the user. It is robust, and could also have been used in the presentation of our type system.

Labels are used to trace the sharing of polytypes. Types could be restricted so that two polytypes with the same label are necessarily equal. This was not required in the present type system, although this property remains valid in all types appearing in a principal derivation of a judgment for which the property is already valid.

The grammar of types can be extended with a sharing construct[1]:

$$\tau ::= \ \ldots \mid (\tau \ \texttt{where} \ \alpha = \tau)$$

Using bindings, any type can always be written such that every label occurs at most once, and thus can be omitted. In fact, in our presentation, sharing of

[1] Alternatively, one could use the binding τ `as` α as in Objective ML, although the binding scope of `as` is less clear and harder to deal with, formally.

types is preserved during type inference. Sharing was just ignored when reading principal solutions from unificands in solved form. The `where` construct allows to read and print all sharing existing in the solved form. Actually, only sharing involving polytypes needs to be printed, and other sharing could still be ignored as before.

For instance, the expression $\lambda x.\,(x : [\sigma])$ has type $[\sigma] \to [\sigma]$, since the two polytypes have different labels, but the expression $\lambda x.\,\texttt{let}\; y = (x : [\sigma])\; \texttt{in}\; x$ has type $(\alpha \to \alpha\; \texttt{where}\; \alpha = \sigma)$.

3 Encodings

In this section, we give encodings in our language for both explicit polymorphism through data-types, and system F. This last encoding is direct, and makes our language an alternative to system F, which allows for more explicit type information than ML, but also for more polymorphism.

Syntactic sugar

It is convenient to allow $\lambda x{:}\tau.\,a$ in expressions. We see such expressions as syntactic sugar for $\lambda x.\,\texttt{let}\; x = (x : \tau)\; \texttt{in}\; a$. The derived typing rule is:

$$\textsc{(Poly-Fun)}\quad \frac{A \oplus (x : \forall FL(\tau_2) \setminus FL(\tau_1).\tau_2) \vdash a : \tau' \qquad (\tau_1 : \tau : \tau_2)}{A \vdash \lambda x{:}\tau.\,a : \tau_1 \to \tau'}$$

The derived reduction is $(\lambda x{:}\tau.\,a)\; v \xrightarrow{\mathsf{Fun}} a\{(v : \tau)/x\}$.

Encoding polymorphic data-type

Previous works have used data types to provide explicit polymorphism [6, 14, 9]. Omitting other aspects that are irrelevant here, all these works amount to an extension of ML with expressions of the form:

$t ::= \alpha \mid t \to t \mid T\; \bar{\alpha}$	Types
$M ::= x \mid M\; M \mid \lambda x.\, M \mid T\; M \mid T^{-1}\; M$	Terms
$\mid \texttt{type}\; T\; \bar{\alpha} = \sigma\; \texttt{in}\; e$	Type declarations

where T ranges over data type symbols. In expressions, T and T^{-1} act as mutually inverse introduction and elimination functions to coerce the higher-order types σ into the simple type $T\; \bar{\alpha}$. For simplicity, we can assume without loss of generality that every type symbol T occurs only once, and we write $(\Lambda\bar{\alpha}.\,\sigma)$ for the type symbol T associated with the definition $\texttt{type}\; T\; \bar{\alpha} = \sigma$.

The translation of types into types of our language is straightforward.

$$\langle\!\langle \alpha \rangle\!\rangle = \alpha \qquad \langle\!\langle t \to t \rangle\!\rangle = \langle\!\langle t \rangle\!\rangle \to \langle\!\langle t \rangle\!\rangle \qquad \langle\!\langle (\Lambda\bar{\alpha}.\,\sigma)\; \bar{\tau} \rangle\!\rangle = [\sigma\{\tau/\bar{\alpha}\}]$$

Note that all type schemes are translated as polytypes with anonymous labels. We translate programs as follows.

$$\langle\!\langle x \rangle\!\rangle = x \qquad \langle\!\langle \lambda x.\, a \rangle\!\rangle = \lambda x.\, \langle\!\langle a \rangle\!\rangle \qquad \langle\!\langle a_1\ a_2 \rangle\!\rangle = \langle\!\langle a \rangle\!\rangle_1\ \langle\!\langle a \rangle\!\rangle_2$$

$$\langle\!\langle (\Lambda\bar{\alpha}.\, \sigma)\ M \rangle\!\rangle = [\langle\!\langle M \rangle\!\rangle : \sigma] \qquad \langle\!\langle (\Lambda\bar{\alpha}.\, \sigma)^{-1}\ M \rangle\!\rangle = \langle M : [\sigma] \rangle$$

Indeed, the pattern $\langle \_ : [\sigma] \rangle$ amounts to the explicit elimination of polymorphism. Since, in the translation, the elimination of polymorphism is always explicit, it can easily be shown that the translation of a well-typed term is always well-typed.

Encoding system *F*

Laüfer and Odersky have shown an encoding of system F into polymorphic data-types [9]. This guarantees by composition that system F can be encoded into semi-explicit polymorphism. We give here a direct encoding of system F, which is much simpler than the encoding into polymorphic data-types.

The types and the terms of system F are

$$\begin{array}{rll} t & ::= \alpha \mid t \to t \mid \forall\alpha.t & \text{Types} \\ M & ::= x \mid M\ M \mid \lambda x{:}t.\, M \mid \Lambda\alpha.\, M \mid M\ t & \text{Terms} \end{array}$$

The translation of types of system F into types of our language is again straightforward:

$$\langle\!\langle \alpha \rangle\!\rangle = \alpha \qquad \langle\!\langle t \to t \rangle\!\rangle = \langle\!\langle t \rangle\!\rangle \to \langle\!\langle t \rangle\!\rangle \qquad \langle\!\langle \forall\alpha.t \rangle\!\rangle = [\forall\alpha.\langle\!\langle t \rangle\!\rangle]$$

The translation $\langle\!\langle \_ \rangle\!\rangle$ is extended to typing environments in an homomorphic way. The translation of typing derivations of terms of system F into terms of our language is given by the following inference rules:

$$\frac{x : t \in A}{A \vdash x : t \Rightarrow (x : \langle\!\langle t \rangle\!\rangle)} \qquad \frac{A \oplus (x : t) \vdash M : t' \Rightarrow a}{A \vdash \lambda x{:}t.\, M : t \to t' \Rightarrow \lambda x.\, a}$$

$$\frac{A \vdash M : t' \to t \Rightarrow a \qquad A \vdash e' : t' \Rightarrow a'}{A \vdash M\ e' : t \Rightarrow a\ a'} \qquad \frac{A \vdash M : t \Rightarrow a \qquad \alpha \notin FV(A)}{A \vdash \Lambda\alpha.\, M : \forall\alpha.t \Rightarrow [a : \forall\alpha.\langle\!\langle t \rangle\!\rangle]}$$

$$\frac{A \vdash M : t' \Rightarrow a}{A \vdash M\ t : t'\{t/\alpha\} \Rightarrow \langle a \rangle}$$

Since the translation rules copy the typing rules of system F, the translation is defined for all well-typed terms. There is no ambiguity and the translation is deterministic.

Lemma 3. *For any term M of system F, if $A \vdash M : t \Rightarrow a$, then $\langle\!\langle A \rangle\!\rangle \vdash a : \langle\!\langle t \rangle\!\rangle$.*

Proof (sketch). The proof is by structural induction on M. The only difficulty is to ensure that when typing $\langle a \rangle$ the polytype $[\sigma]^{\epsilon}$ of a is always anonymous. Since the translation of all variables is a type constraint, which renames all labels of its type, it can easily be shown that the type of an expression never shares any label with the typing environment.

If we choose for system F the semantics where type abstraction does not stop evaluation (*i.e.* $\Lambda\alpha. E$ is an evaluation context whenever E is), then the translation preserves the semantics in a strong sense (reduction steps of a term can be mapped to the reduction of the translated term). Another semantics would need easy adjustment, either of the translation or of the semantics of our system.

Using the extended syntax $\lambda x{:}\tau.\, a$, we could replace the two first rules by:

$$\frac{x : t \in A}{A \vdash x : t \Rightarrow x} \qquad \frac{A \oplus (x : t) \vdash M : t' \Rightarrow a}{A \vdash \lambda x{:}t.\, M : t \to t' \Rightarrow \lambda x{:}\, \langle\!\langle t \rangle\!\rangle .\, a}$$

This allows for a closer comparison between system F and our language. Let us compare a term M and its translation a. The type information on lambda abstractions is the same in both terms. The type information at the elimination of polymorphism is always omitted in a. The counterpart is that type information at the introduction of polymorphism is richer in a, since it must give the full type of the expression, not just the type variable that is abstracted.

Our language is also more flexible: annotations of abstractions are not mandatory and, in particular, ML programs do not require any explicit type information at all; multiple abstractions can also be introduced simultaneously, as in $[a : \forall \alpha_1, \alpha_2.\tau]$. Since type application is explicit in system F, the expression $\Lambda\alpha_1, \alpha_2.\, M$ would be ambiguous; thus it is not allowed.

The simplicity of our encoding compared to the encoding into polymorphic data-types is permitted by the introduction of polytypes as first-class types, and does not rely on the inference of polytypes at the elimination. If we leave the elimination of polymorphism fully explicit, we could keep first-class polytypes but omit all labels in polytypes. We would obtain a weaker but simpler proposal that would extend ML and be as powerful as system F, but more verbose.

4 Extensions to the core language

In this section, we describe two independent extensions to the core language. First, we study the restriction of polymorphism to values, which is commonly accepted as the best solution for keeping type-soundness in the presence of side effects. Then, we extend rule Use to allow a more uniform treatment of monomorphic and polymorphic polytypes; polymorphic methods in Objective ML are an important application.

4.1 Value-only polymorphism

For impure functional programming languages, value-only polymorphism has become the standard way to handle the ubiquity of side-effects. It is based on a very simple idea —if an expression is *expansive*, *i.e.* its evaluation may produce side-effects, then its type should not be polymorphic [17].

This is usually incorporated by restricting the Gen rule to a class of expressions b, called non-expansive, composed of variables and functions. Equivalently, this restriction can be put on the Let rule: both ways give exactly the same canonical derivations in the core language. We actually prefer this way, since it still allows us to generalize ϵ before the Use rule, which is needed and correct.

Thus, we replace rules Poly and Let by the following four rules, each rule being split in its expansive and non-expansive versions.

(Poly-V)
$$\frac{A \vdash b : \sigma_1 \qquad (\sigma_1 : \sigma : \sigma_2)}{A \vdash [b : \sigma] : [\sigma_2]^\epsilon}$$

(Poly-E)
$$\frac{A \vdash a : \tau_1 \qquad (\tau_1 : \tau : \tau_2)}{A \vdash [a : \tau] : [\tau_2]^\epsilon}$$

(Let-V)
$$\frac{A \vdash b : \varsigma \qquad A \oplus x : \varsigma \vdash a : \tau}{A \vdash \texttt{let}\ x = b\ \texttt{in}\ a : \tau}$$

(Let-E)
$$\frac{A \vdash a_1 : \tau' \qquad A \oplus x : \tau' \vdash a_2 : \tau}{A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau}$$

The class of non-expansive expressions can be refined, provided the evaluation cannot produce side-effects and preserves non-expansiveness. For instance, in ML, we can consider let-bindings of non-expansive expressions in non-expansive expressions as non-expansive. In our calculus, type annotations are also non-expansive. More generally, any expression where every application is protected (*i.e.* appears) under an abstraction is non-expansive (creation of mutable data-structure would be the application of a primitive):

$$b ::= x \mid \lambda x.\, a \mid \texttt{let}\ x = b\ \texttt{in}\ b \mid (b : \tau) \mid [b : \sigma] \mid \langle b \rangle$$

This system works perfectly, and all properties are preserved.

However, it seems too weak in practice. Since we use polymorphism of ϵ's to denote confirmation of polytypes, as soon as we let-bind an expansive expression, all its ϵ's become monomorphic, and all its polytypes weak. For instance, the following program is not typable, because labels in the type of the binding occurrence of g cannot be generalized.

$$\texttt{let}\ f = [\lambda x.\, x : \forall \alpha.\alpha \to \alpha]\ \texttt{in}\ \texttt{let}\ g = (\lambda x.\, x)\ f\ \texttt{in}\ \langle g \rangle\ g$$

When ML polymorphism is restricted to values, the result of an application is monomorphic (here, the result of applying $\lambda x.\, x$ to f). Traditionally, the typical situation when a polymorphic result is restricted to be monomorphic is partial application. Polymorphism is there easily recovered by η-expansion. However, the same problem appears when objects are represented as records of methods, with no possibility of η-expansion. In our core language, the only way to recover at least explicit polymorphism in such a case is to annotate the use of let-bound variables with their own types:

$\texttt{let}\ f = [\lambda x.\, x : \forall \alpha.\alpha \to \alpha]\ \texttt{in}$
$\texttt{let}\ g = (\lambda x.\, x)\ f\ \texttt{in}\ \langle g : [\forall \alpha.\alpha \to \alpha] \rangle\ g$

In practice, with objects, this means recalling explicit polymorphism information at each method invocation. The strength of our system being its ability to omit such information, this limitation would significantly reduce its interest.

One might think that allowing quantification on ϵ in LET-E, *i.e.* write $\forall\bar{\epsilon}.\tau'$ in place of τ', is harmless. Indeed, ϵ's polymorphism does not allow type mismatches like α's polymorphism would: verifying identity of type schemes is done separately. However, this rule would break principal types. Consider, for instance, the following expression:

$$\texttt{let}\ x = \texttt{id}\ []\ \texttt{in}\ \texttt{let}\ y = \langle \texttt{hd}\ x \rangle\ \texttt{in}\ x$$

It can be assigned type $[\sigma]^\epsilon$ `list` for any type scheme σ. Since type schemes of polytypes are not ordered, there is no principal type for this expression.

This problem is pathological, but not anecdotical. It can be solved by using principal judgments. That is, we replace LET-V and LET-E by the following restricted rules. $A \vdash^\star a : \varsigma$ means that ς is the most general scheme for a under assumptions A.

(LET-V*)
$$\frac{A \vdash^\star b : \varsigma \qquad A \oplus x : \varsigma \vdash a : \tau}{A \vdash \texttt{let}\ x = b\ \texttt{in}\ a : \tau}$$

(LET-E*)
$$\frac{A \vdash^\star a_1 : \forall\bar{\epsilon}\bar{\alpha}.\tau' \qquad a_1 \not\equiv b \qquad A \oplus x : (\forall\bar{\epsilon}.\tau')\{\bar{\tau}/\bar{\alpha}\} \vdash a_2 : \tau}{A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau}$$

This restriction to principal judgments is not new: it has already been used for the typing of dynamics in ML [7] or value-only polymorphism in Standard ML'96 to disallow monomorphic variables at toplevel. In the former case, the program $\lambda x.(\texttt{dynamics}\ x)$ is rejected because, in the principal judgment $x : \alpha \vdash x : \alpha$, some variable of the type of x occurs free in the context. Similarly, $(\lambda x. x)\ (\lambda x. x)$ fails to type in Standard ML'96 because the non generalizable variable α is free in the principal type $\alpha \to \alpha$. In both cases, a non principal judgment obtained by choosing `int` for α would be correct, were it not for the principality condition.

Still, we do not consider this solution as fully satisfactory, and we view it as an example of the difficulties inherent to value-only polymorphism.

4.2 An extended language

In this section we show how the core language can be used to provide polymorphic methods in Objective ML[2] [15]. Polymorphic methods are useful in parameterized classes. Indirectly, they may also reduce the need for explicit coercions.

[2] The examples of objects and classes given below are rather intuitive, and could be translated in other class-based object-oriented languages; the reader may refer to [15] for a formal presentation of Objective ML.

While Objective ML has parametric classes, it does not allow methods to be polymorphic. For instance, the following class definition fails to type.

```
let α collection = class (l)
    val contents = l
    meth mem = λx. mem x contents
    meth fold : (β → α → β) → β → β
              = λf.λx. fold_left f x contents
end
```

The reason is that variable β is free in the type for method `fold`, but is not bound to a class parameter. The solution is to have the method `fold` be polymorphic in β. With polytypes, we can write

```
    meth fold = [λf.λx. fold_left f x contents
                   : ∀β. (β → α → β) → β → β]
```

Still, we have to distinguish between polymorphic and monomorphic methods, in particular when we send a message to the object. The aim of the remainder of this section is to make use of polymorphic and monomorphic methods similar, and more generally the use of polymorphic methods smoother.

The first step is to give all methods polytypes. This is easily done by wrapping monomorphic methods in polytypes. For instance,

```
    meth mem = [λx. mem x l : α]
```

However, we still want to be able to use monomorphic methods without type annotations. There is a small but very convenient extension of the core language thats solves this problem. We add a new typing rule USE-M:

$$\text{(USE-M)}\quad \frac{A \vdash a : [\tau]^{\epsilon}}{A \vdash \langle a \rangle : \tau}$$

As opposed to rule USE, this one allows ϵ to appear in A. Inference problems are solved by forcing the polytype to be monomorphic.

Both rules USE and USE-M apply when ϵ is anonymous and the polytype is monomorphic, but they produce the same derivation. If either ϵ is free in A, or the polytype is polymorphic, then only one of the two rules may be used. As a result, principal types are preserved. The type inference algorithm can be modified as shown in figure 5. Subject reduction property is also preserved.

The expression `(λx.λy. ⟨x#mem⟩ y)` is then typable with principal type `⟨mem : [α → β]; ..⟩ → α → β`. Since all methods are now given polytypes, we will change our notations (the new notations are given in term of the old ones): in types, we now write $m : \sigma$ for $m : [\sigma]$; in expressions, we now write $m : \sigma = a$ for $m = [a : \sigma]$, $m = a$ for $m = [a : \alpha]$ and $a\#m$ for $\langle a\#m \rangle$. With the new notations, the collection example is written:

```
let α collection = class (l)
    val contents = l
```

```
        meth mem = λx. mem x contents
        meth fold : ∀β. (β → α → β) → β → β
                  = λf.λx. fold_left f x contents
    end;;
```
value collection : class α (α list)
meth mem : α → bool
meth fold : ∀β. (β → α → β) → β → β
end

A monomorphic method is used exactly as before.

```
    let coll_mem c x = c#mem x
```
coll_mem : ⟨mem : α → β; ..⟩ → α → β

However, when polymorphic methods are used under abstractions, the type of the object should be provided as an annotation,

```
    let simple_and_double (c : α collection) =
        let l1 = c#fold (λx.λy. x::y) [] in
        let l2 = c#fold (λx.λy. (x,x)::y) [] in
        (l1, l2);;
```
*simple_and_double : α collection → (α list * (α * α) list)*

Since the method `fold` is used with two different types, this example could not be typed without first-class polymorphism.

Polymorphic methods also appear to be useful to limit the need for explicit coercions. In Objective ML, coercions are explicit. For instance suppose that objects of class point have the interface ⟨`x : int`; `y : int`⟩, and that we want to define a class circle with a method giving the distance from the circle to a point.

```
    let circle = class (x,y,r)  ...
        meth distance = λp:point. ...
    end;;
```
*value circle : class (int * int * int) ...*
meth distance : point → float
end

Given a point `p` and a circle `c`, we can get their distance by `c#distance p`. However, an object `cp` of a class `color_point` where `color_point` is a subtype of `point` (*e.g.* its interface is ⟨`x : int`; `y : int`; `color : color`⟩) needs to be explicitly coerced to `point` before its distance to the circle can be computed:

```
    c#distance (cp : color_point :> point)
```

This coercion could be avoided if `distance` were a toplevel function rather than a method:

```
    let distance c p = c#distance (p :> point);;
```
value distance : ⟨distance : point → α ; ..⟩ → #point → α

The type expression `#point` represents any subtype of `point`. Actually, it is an abbreviation for the type ⟨`x : int`; `y : int`; ρ⟩. Here, `#point` contains a hidden row variable that is polymorphic in the function `distance`. This allows different applications to use different instances of the generic row variable and thus to accept different objects all matching the type of points.

Explicit polymorphism allows to recover the same power inside methods:

```
    meth distance : ∀α:#point. α → float = λp. ...
```

Then, `c#distance cp` is typable just by instantiation of these row variables, without explicit coercion. Of course, this also means that we have to know that `c` is a circle before using method `distance`, like would happen in more classical object-oriented type systems. There is a choice here between using explicit coercions and giving more type information. The advantage of type information is that it occurs at more convenient places, *i.e.* at method definitions and invocation of a method on an object of unknown type. On the opposite, explicit coercions must be repeated at each invocation of a method even when all types are known.

Related Work

Full type inference of polymorphic types is undecidable [16]. Several works have studied the problem of partial type inference in system F.

Some implementations of languages based on system F relieve the user from the burden of writing all types down. In Cardelli's implementation of the language Fun [1] some polymorphic polytypes may be marked as implicit (actually their variables are marked) and automatically instantiated when used, or marked to stay polymorphic. This mechanism turns out to be quite effective in inferring type applications. However, types of abstracted values are never inferred. Thus, the expression $\lambda x.x$ cannot be typed without providing a type on x, which shows that this is not an extension of ML. Pierce and Turner have extended this partial inference mechanism to $F^{\omega}_{<:}$ in the design of the language Pict [12]. By default they also assign "unification variables" to parameters of functions with no type annotations. Their solution requires surprisingly little type information in practice, especially in the absence of subtyping. Still, as for Cardelli's solution, it is quite difficult to know exactly the set of well-typed programs, since the description is only algorithmic.

A different approach is taken by Pfenning [10]. Instead of providing type annotations on lambda's he indicates possible type applications (this corresponds to the notation $\langle \_ \rangle$ in our language). Then, he shows that partial type inference in system F corresponds to second-order unification and is thus undecidable [11]. As ours, his solution is an extension of ML. It is also more powerful; the price is the loss of principal types and decidability of type inference. As explained in the introduction, we designed our system never to guess higher-order types.

Kfoury and Wells show that type inference could be done for the rank-2 fragment of system F [5]. However, they do not have a notion of principal types. It is also unclear how partial type information could be added.

In [9], Läufer and Odersky actually present two different mechanisms. First, as we explained in the introduction they add higher-order polymorphism with fully explicit introduction and elimination. As we have seen, our framework subsumes theirs. They also introduce another mechanism that allows annotations of abstractions by type schemes as in $\lambda x{:}\,\sigma.\,x$ together with a type containment relation on type schemes similar to the one of Mitchell [8]. Type schemes may be of the form $\forall\alpha.\sigma_1 \rightarrow \sigma_2$, where σ_i are type scheme themselves. However, universal variables such as α can only be substituted by simple types. Thus, the only way to apply a function of type $\forall\alpha.\alpha \rightarrow \alpha$ to a polymorphic value remains to embed the argument inside an explicitly defined polytype.

In [3], Duggan proposes an extension to ML with objects and polymorphic methods. His solution heavily relies on the use of kinds and type annotations. These are carried by method names that must be declared before being used. In this regard, his solution is similar to fully explicit polymorphism both at introduction and elimination, as the one of Läufer and Odersky. His use of recursive kinds allows some programs that cannot be typed in our proposal (section 4). However, this is due to a different interpretation of object types rather than a stronger treatment of polymorphism.

Conclusion

Our extension of ML allows a more convenient use of polymorphic types. Polytypes are created with explicit type annotations, and can be used without specifying their types, except under abstraction. This is particularly useful in Objective ML to allow methods to be polymorphic.

Our solution is practical, since it can be used with the value-only polymorphism restriction, that is, in the presence of side-effects. We propose two options. The first, standard solution preserves all fundamental properties, but is weaker. The second solution is stronger, covers all useful cases, and does not present any new limitations. However, it keeps principal types only under some restriction. This is insignificant in practice, but reveals the limitation of value-only polymorphism.

Our approach is to keep type inference first-order, since we believe that this is sufficient in practice. Still, we allow the explicit introduction of higher-order polymorphism and its smooth interaction with ML polymorphism. As future work we would like to present our type system closer to the framework of partial type inference for second-order lambda-calculus.

The possible application to a similar simplification of first-class existential polymorphism in ML also remains to be investigated.

References

1. Luca Cardelli. An implementation of FSub. Research Report 97, Digital Equipment Corporation Systems Research Center, 1993.
2. 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.

3. Dominic Duggan. Polymorphic methods with self types for ML-like languages. Technical report cs-95-03, University of Waterloo, 1995.
4. J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. MIT-Press, Cambridge (MA, USA), 1991.
5. A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ-calculus. In *Proceedings of the ACM Conference on Lisp and functional programming*, pages 196–207, Orlando, Florida, June 1994.
6. 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.
7. Xavier Leroy and Michel Mauny. Dynamics in ML. In John Hughes, editor, *Conference on Functional Programming and Computer Achitecture*, volume 523 of *Lecture Notes in Computer Science*, pages 406–426. Springer-Verlag, 1991.
8. John C. Mitchell. Polymorphic type inference and containment. In *Proceedings of the International Symposium on Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 257–278, Sophia-Antipolis, France, June 1984. Springer-Verlag. Full version in *Information and Computation*, 76(2/3):211–249, 1988. Reprinted in *Logical Foundations of Functional Programming,* ed. G. Huet, pages 153–194, Addison-Wesley, 1990.
9. 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.
10. 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.
11. 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.
12. Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical report, Computer Science Department, Indiana University, 1997.
13. Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992.
14. 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.
15. Didier Rémy and Jerôme Vouillon. Objective ML: A simple object-oriented extension to ML. In *Proceedings of the 24th ACM Conference on Principles of Programming Languages*, pages 40–53. ACM Press, January 1997.
16. 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, Paris, France, July 1994.
17. Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical Report 93–200, Rice University, February 1993.

A Proofs of main theorems

Proof of type soundness for the core language

Lemma 4 (Term substitution). *If $A \oplus x : \sigma_2 \vdash a : \sigma_1$ and $A \vdash v : \sigma_2$ hold, then $A \vdash a\{v/x\} : \sigma_1$ also holds.*

Proof. The proof is an easy induction on the structure of v.

Theorem 4 (Subject reduction). *Reduction preserves typings,* i.e. *if $a_1 \longrightarrow a_2$, then $a_1 \subset a_2$.*

Proof. We show that every rule in the definition of $\longrightarrow$ is satisfied by the relation $\subset$. Since $\longrightarrow$ is the smallest relation verifying those rules, then $\subset$ must be a super-relation of $\longrightarrow$. All cases are independent. In each case, we assume that $A \vdash a_1 : \sigma$ (1) and that $a_1 \longrightarrow a_2$, (the structure of a_1 depending on the case) and we show that $A \vdash a_2 : \sigma$ (2).

We first assume that the derivation does not end with a rule Gen If the derivation ends with a rule Gen, then it is of the form:

$$\frac{A \vdash a : \varsigma}{A \vdash a : \forall \bar{\xi}.\varsigma}\ (\textsc{Gen}^*)$$

and there is a derivation of (1) that does not end with a rule Gen. Thus we have $A \vdash a_2 : \sigma$ and (2) follows by the same application of rule Gen*.

Case Fun *and* Let*:* This is a straightforward application of term-substitution lemma.

Case Use*:* A canonical derivation of (1) ends with

$$\cfrac{\cfrac{\cfrac{A \vdash a : \sigma_1 \qquad (\sigma_1 : \sigma_0 : \sigma_2)}{A \vdash [a : \sigma_0] : [\sigma_2]^\epsilon}\ (\textsc{Poly})}{A \vdash [a : \sigma_0] : \forall \epsilon.[\sigma_2]^\epsilon}\ (\textsc{Gen})}{A \vdash \langle [a : \sigma_0] \rangle : \sigma_2}\ (\textsc{Use})$$

The type schemes σ_1, σ_0, and σ_2 are of the form $\forall \bar{\alpha}.\tau_1$, $\forall \bar{\alpha}.\tau_0$, and $\forall \bar{\alpha}.\tau_2$, and such that $(\tau_1 : \tau_0 : \tau_2)$. Choosing $\bar{\alpha}$ such that they do not occur free in A, we can contract this derivation into

$$\cfrac{\cfrac{\cfrac{A \vdash a : \sigma_1}{A \vdash a : \tau_1}\ (\textsc{Inst}^*) \qquad (\tau_1 : \tau_0 : \tau_2)}{A \vdash (a : \tau_0) : \tau_2}\ (\textsc{Ann})}{A \vdash (a : \tau_0) : \sigma_2}\ (\textsc{Gen}^*)$$

Case TFUN*:* A canonical derivation of $A \vdash a_1 : \sigma$ ends with

$$\dfrac{\dfrac{A \vdash v_1 : \tau_2' \to \tau_1' \qquad (\tau_2' \to \tau_1' : \tau_1 \to \tau_2 : \tau_2'' \to \tau_1'')\ (3)}{A \vdash (v_1 : \tau_2 \to \tau_1) : \tau_2'' \to \tau_1''}\,(\textsc{Ann}) \qquad A \vdash v_2 : \tau_2''}{A \vdash (v_1 : \tau_2 \to t_1)\ v_2 : \tau_1''}\,(\textsc{App})$$

Since the relation (3) implies both $(\tau_2'' : \tau_2 : \tau_2')$ and $(\tau_1' : \tau_1 : \tau_1'')$, we can build the derivation:

$$\dfrac{\dfrac{A \vdash v_1 : \tau_2' \to \tau_1' \qquad \dfrac{A \vdash v_2 : \tau_2'' \qquad (\tau_2'' : \tau_2 : \tau_2')}{A \vdash (v_2 : \tau_2) : \tau_2'}\,(\textsc{Ann})}{A \vdash v_1\ (v_2 : \tau_2) : \tau_1'}\,(\textsc{App}) \qquad (\tau_1' : \tau_1 : \tau_1'')}{A \vdash (a\ (b : \tau_1) : \tau_2) : \tau_1''}\,(\textsc{Ann})$$

Case TPOL*:* The last derivation of (1) ends with:

$$\dfrac{\dfrac{A \vdash v : \sigma_1'\ (3) \qquad (\sigma_1' : \sigma_1 : \sigma_1'')\ (4)}{A \vdash [v : \sigma_1] : [\sigma_1'']^{\epsilon'}}\,(\textsc{Poly}) \qquad ([\sigma_1'']^{\epsilon_1} : [\sigma_2]^{\epsilon_2} : [\sigma_3]^{\epsilon_3})\ (5)}{A \vdash ([v : \sigma_1] : [\sigma_2]^{\epsilon_2}) : [\sigma_3]^{\epsilon_3}}\,(\textsc{Ann})$$

Let $\forall \xi.\tau_1$ be σ_1. From (4), we know that we can write σ_1' and σ_1'' as $\forall \xi.\sigma_1'$ and $\forall \xi.\sigma_1''$. Moreover, we have $(\tau_1' : \tau_1 : \tau_1'')$ (6). From (5), We also get $(\sigma_1'' : \sigma_2 : \sigma_3)$ (7). Thus, we have

$$\dfrac{\dfrac{\dfrac{\dfrac{(3)}{A \vdash v : \tau_1'}\,(\textsc{Inst*}) \qquad (6)}{A \vdash (v : \tau_1) : \tau_1''}\,(\textsc{Ann})}{A \vdash (v : \tau_1) : \sigma_1''}\,(\textsc{Gen*}) \qquad (7)}{(2)}\,(\textsc{Poly})$$

Case TVAR*:* Annotating with a variable does nothing.

Theorem 5 (Canonical forms). *Irreducible programs that are well-typed in the empty environment are values.*

Proof. We first relate the shape of types and the shape of values. Let v be a value of type τ. By considering the possible canonical derivations, we can show that:

- if v is a poly expression, or a poly expression with a type constraint, then τ is a polytype;
- otherwise, v is of the form w and τ is a functional type.

Since polytype and functional types are incompatible, we can invert the property:

- if τ is a polytype, then v is a poly expression.
- otherwise, τ is a functional type, and v is of he form w.

Then we can easily prove the theorem: we consider a program a that is well-typed in the empty environment, and that cannot be reduced. We easily show by induction on the structure of a that it is a value.

Proof of the principal type property

Theorem 6. *Given a typing problem $(A \triangleright a : \tau)$ there exists a principal solution, which is computed by the set of rules described in figures 2, 3 and 4.*

Proof. We first show the soundness and completeness of each rewriting rule:

Cases VAR*,* FUN*,* APP*, and* LET*:* are as in ML.

Case ANN*:* The case ANN is not special since the construct $(\_ : \tau)$ could be treated as the application of a primitive.

Case POLY*:* We assume that all the conditions of the first two lines are satisfied. We write σ_i for $\sigma\{\bar\alpha_1\bar\epsilon_i/\bar\alpha_0\bar\epsilon_0\}$.

Soundness: If $\bar\alpha \cap \mathit{dom}(\theta) \cup FV(\mathit{codom}(\theta)) = \emptyset$, we have $\theta(A) \vdash a : \theta(\sigma_1)$ by generalization of $\bar\alpha$ in the judgement $\theta(A) \vdash a : \theta(\tau_1\{\bar\epsilon_1/\bar\epsilon_0\})$. Since $(\theta(\sigma_1) : \sigma : \theta(\sigma_2))$, we have $\theta(A) \vdash [a : \sigma] : \theta([\sigma_2]^\epsilon)$ That is, θ is a solution of $A \vdash [a : \sigma] : [\sigma_2]^\epsilon$. Thus, a solution of $\theta \wedge \tau = [\sigma_2]^\epsilon$ is a solution of $A \vdash [a : \sigma] : \tau$. Moreover, no variable of $\bar\epsilon_1, \bar\epsilon_2, \epsilon, \bar\alpha_1$ appears in A or τ.

Completeness: Let us assume that θ' is a solution of $A \triangleright [a : \sigma] : \tau$. The canonical derivation of $\theta'(A) \vdash [a : \sigma] : \theta'(\tau)$ must end with rule POLY. Thus, $\theta'(A) \vdash a : \theta'(\sigma_1)$ (1) for some extension of θ' to $\bar\alpha_1$, and $\theta'(\tau)$ is of the form $[\theta'(\sigma_2)]^{\epsilon'}$, that is θ' is a solution of $\tau \doteq [\sigma_2]^{\epsilon'}$ (2).

From (1), we have $\theta'(A) \vdash a : \theta'(\tau_1\{\bar\epsilon_1/\bar\epsilon_0\})$. Since θ' is a solution of $A \vdash a : (\tau_1\{\bar\epsilon_1/\bar\epsilon_0\})$. it is also a solution of $\exists\,\bar\xi.\,\theta$. That is, it can be extended on $\bar\xi$ into a solution of θ. Together with (2), this shows the completeness of the first case. No variable of $\bar\alpha$ appears in $\mathit{dom}(\theta')$ nor in $\mathit{codom}(\theta')$, otherwise (1) could not hold. Therefore, no variable of $\bar\alpha$ can either appear in $\mathit{dom}(\theta)$ nor in $\mathit{codom}(\theta)$. This shows the completeness of the failure case.

Case USE*:* We assume that all the conditions of the first line are satisfied .

Soundness: If $\theta(\alpha) = [\forall\bar\alpha'.\tau']^\epsilon$ and $\epsilon \notin FL(\theta(A))$ then an extension of θ such that $\theta(\tau) = \theta(\tau')$ is clearly a solution of $A \triangleright \langle a\rangle : \tau$. If $\theta(\alpha) = \alpha'$ and $\alpha' \notin FV(\theta(A))$ then from $\theta(A) \vdash a : \alpha'$ we deduce $\theta(A) \vdash a : [\tau]^\epsilon$ for some ϵ not in $FV(\theta(A))$. By generalization of ϵ and rule USE, we get $\theta(A) \vdash \langle a\rangle : \theta(\tau)$. The substitution θ is thus a solution of $A \triangleright \langle a\rangle : \tau$.

Completeness: Let us assume that θ' is a solution of $A \vdash \langle a\rangle : \tau$. The canonical derivation of $\theta'(A) \vdash \langle a\rangle : \theta'(\tau)$ must end with rule USE. Thus, we must have $\theta'(A) \triangleright a : [\sigma]^{\epsilon'}$ for some ϵ' that does not appear in $\theta'(A)$ and some type scheme σ of which $\theta'(\tau)$ is an instance. Since $\exists\,\bar\xi.\,\theta$ is a principal solution of $A \triangleright a : \alpha$, θ' can be extended on $\bar\xi$ into a solution of $\theta \wedge \theta(\alpha) \doteq [\sigma]^{\epsilon'}$ (1).

Therefore $\theta(\alpha)$ cannot be an arrow type. If it is a variable, then it cannot belong to $\theta(A)$, otherwise ϵ' would belong to $\theta'(A)$. Hence, together with (1) the completeness of the second and third cases.

If $\theta(\alpha) = [\forall\bar\alpha'.\tau']^\epsilon$ then ϵ cannot belong to $FL(\theta(A))$, otherwise ϵ' would belong to $\theta'(A)$. Since θ' is a solution of $[\sigma]^{\epsilon'} \doteq [\forall\bar\alpha'.\tau']^\epsilon$, it is also a solution of

$\sigma = \forall \bar{\alpha}'.\tau$. Since $\theta'(\tau)$ is an instance of σ, it is an instance of $\forall \bar{\alpha}'.\tau$. Thus θ' can be extended on $\bar{\alpha}'$ into a solution of $\tau = \tau'$. Together with (1), θ' is a solution of $\theta \wedge \tau = \tau'$.

Termination: We now show that applying the rules in any order always terminates, with a unification problem in solved form.

Each rule of the algorithm decreases the lexicographic ordering composed of successively the total size of program components, the total size of types, the number of polymorphic constraints, the number of multi-equations, and the number of renaming problems.

Moreover, unification problems that cannot be reduced are in solved form, since the side conditions of type-inference rules can never fail. Thus, a general unification problem in normal form is a unification problem in normal form, *i.e.* it is in solved form.

```
VAR
    if $\forall\bar{\xi}.\tau' = A(x)$ and $\bar{\xi} \cap FV(\tau) = \emptyset$ then
    $A \triangleright x : \tau \Rrightarrow \exists\,\bar{\xi}.\,\tau \doteq \tau'$
FUN
    if $\alpha_1, \alpha_2 \notin FV(A) \cup FV(\tau)$ then
    $A \triangleright \lambda x.\,a : \tau \Rrightarrow \exists\,\alpha_1, \alpha_2.\,(A \oplus x : \alpha_1 \triangleright a : \alpha_2) \wedge \tau \doteq \alpha_1 \to \alpha_2$
APP
    if $\alpha \notin FV(A) \cup FV(\tau)$ then
    $A \triangleright a_1\ a_2 : \tau \Rrightarrow \exists\,\alpha.\,(A \triangleright a_1 : \alpha \to \tau) \wedge (A \triangleright a_2 : \alpha)$
LET
    if $\alpha \notin FV(A)$ and $A \triangleright a_1 : \alpha \Rrightarrow \exists\,\bar{\xi}.\,\theta$ then
    $A \triangleright \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau \Rrightarrow \exists\,\bar{\xi}, \alpha.\,\theta \wedge A \oplus x : Gen\ (\theta(\alpha), \theta(A)) \triangleright a_2 : \tau$
ANN
    if $\bar{\epsilon}_0 = FL(\tau_0)$ and $\bar{\epsilon}_1$ and $\bar{\epsilon}_2$ are disjoint copies of $\bar{\epsilon}_0$ outside of $A$ and $\tau$
     and $\bar{\alpha}_0 = FV(\tau_0)$ and $\bar{\alpha}_1$ is a copy of $\bar{\alpha}_0$ outside of $A$ and $\tau$
     and $\tau_1 = \tau_0\{\bar{\alpha}_1/\bar{\alpha}_0\}$ then
        $A \triangleright (a : \tau_0) : \tau \Rrightarrow \exists\,\bar{\epsilon}_1, \bar{\epsilon}_2, \bar{\alpha}_1.\,A \triangleright a : \tau_1\{\bar{\epsilon}_1/\bar{\epsilon}_0\} \wedge \tau \doteq \tau_1\{\bar{\epsilon}_2/\bar{\epsilon}_0\}$
POLY
    if $\bar{\epsilon}_0 = FL(\tau_0)$ and $\bar{\epsilon}_1$ and $\bar{\epsilon}_2$ are disjoint copies of $\bar{\epsilon}_0$ outside of $A$ and $\tau$
     and $\bar{\alpha}_0 = FV(\tau_0)$ and $\bar{\alpha}_1$ is a copy of $\bar{\alpha}_0$ outside of $A$ and $\tau$
     and $\tau_1 = \tau_0\{\bar{\alpha}_1/\bar{\alpha}_0\}$ and $\sigma = \forall\bar{\alpha}.\tau_0$ and $\bar{\alpha} \cap FV(A) = \emptyset$
     and $A \triangleright a : \tau_1\{\bar{\epsilon}_1/\bar{\epsilon}_0\} \Rrightarrow \exists\,\xi.\,\theta$ then
        if $\bar{\alpha} \cap (dom\,(\theta) \cup FV(codom\,(\theta))) = \emptyset$ then
            $A \triangleright [a : \sigma] : \tau \Rrightarrow \exists\,\bar{\xi}, \bar{\epsilon}_1, \bar{\epsilon}_2, \bar{\alpha}_1, \epsilon.\,\theta \wedge \tau \doteq [\forall\bar{\alpha}.\tau_1\{\bar{\epsilon}_2/\bar{\epsilon}_0\}]^{\epsilon}$
        else $A \triangleright [a : \sigma] : \tau \Rrightarrow \bot$
USE
    if $\alpha \notin FV(A)$ and $A \triangleright a : \alpha \Rrightarrow \exists\,\bar{\xi}.\,\theta$ then
        if $\theta(\alpha) = [\forall\bar{\alpha}'.\tau']^{\epsilon}$ and $\epsilon \notin FL(\theta(A))$ then
            $A \triangleright \langle a \rangle : \tau \Rrightarrow \exists\,\bar{\xi}, \alpha, \bar{\alpha}'.\,\theta \wedge \tau' \doteq \tau$
        else if $\theta(\alpha) = \alpha'$ and $\alpha' \notin FV(\theta(A))$ then
            $A \triangleright \langle a \rangle : \tau \Rrightarrow \exists\,\bar{\xi}, \alpha.\,\theta$
        else $A \triangleright \langle a \rangle : \tau \Rrightarrow \bot$
```

Fig. 4. Type inference algorithm

```
USE
    if $\alpha \notin FV(A)$ and $A \triangleright a : \alpha \Rrightarrow \exists\,\bar{\xi}.\,\theta$ then
        if $\theta(\alpha) = [\forall\bar{\alpha}'.\tau']^{\epsilon}$ and $\epsilon \notin FL(\theta(A))$ then
            $A \triangleright \langle a \rangle : \tau \Rrightarrow \exists\,\bar{\xi}, \alpha, \bar{\alpha}'.\,\theta \wedge \tau' = \tau$
        else if $\theta(\alpha) = \alpha'$ and $\alpha' \notin FV(\theta(A))$ then
            $A \triangleright \langle a \rangle : \tau \Rrightarrow \exists\,\bar{\xi}, \alpha.\,\theta$
        else if $\epsilon' \notin FV(A)$ then $A \triangleright \langle a \rangle : \tau \Rrightarrow \exists\,\bar{\xi}, \epsilon', \alpha.\,\theta \wedge \alpha \doteq [\tau]^{\epsilon'}$
```

Fig. 5. Type inference rule for use of monomorphic polytypes