Information and Computation **155**, 13400169 (1999)
Article ID inco.1999.2830, available online at http://www.idealibrary.com

Semi-Explicit First-Class Polymorphism for ML†

Jacques Garrigue

Kyoto University Research Institute for Mathematical Sciences, Kitashirakawa-Oiwakecho, Sakyo-ku, Kyoto 606-01, Japan

and

Didier Rémy

INRIA-Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France

We propose a modest conservative extension to ML that allows semi-explicit first-class polymorphism while preserving the essential properties of type inference. In our proposal, the introduction of polymorphic types is fully explicit, that is, both introduction points and exact polymorphic types are to be specified. However, the elimination of polymorphic types is semi-implicit: only elimination points are to be specified as polymorphic types themselves are inferred. This extension is particularly useful in Objective ML where polymorphism replaces subtyping. ©1999 Academic Press

INTRODUCTION

The success of the ML language is due to its combination of several attractive features. Undoubtedly, the polymorphism of ML [Damas and Milner, 1982] —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, returns a principal type.

Based on 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.

† A preliminary version of this paper has been presented at the "Third International Symposium on Theoretical Aspects of Computer Software [Garrigue and Rémy, 1997]."

There are of course cases where one would like to have first-class polymorphism, as in system F. ML allows for polymorphic definitions, but abstractions can only be monomorphic. Traditionally, ML polymorphism is used for definitions of first-class 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 occurrences 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 parameterized by the type of the output. For instance, a function `fold` with the ML type $\forall \beta, \alpha.\ \beta\ \mathtt{list} \to (\beta \to \alpha \to \alpha) \to \alpha \to \alpha$ should become a method for container objects, of type $\forall \alpha.\ (\tau \to \alpha \to \alpha) \to \alpha \to \alpha$ where τ is the type of the elements of the container. The extension of ML with first-class abstract types [Läufer and Odersky, 1994, Rémy, 1994] also requires first-class polymorphic functions: for instance, an expression such as $\lambda f.\mathtt{open}\ x\ \mathtt{as}\ y\ \mathtt{in}\ f\ y$ can only be typed if the argument f is polymorphic in its argument, so that the abstract representation of y is not revealed outside the scope of the open construct. First-class polymorphism seems to be also useful in Haskell to enable the composition of monads.

First-class polymorphic values have been proposed in [Rémy, 1994, Odersky and Läufer, 1996] based on ideas developed in [Läufer and Odersky, 1994]. After de-sugaring, all these proposals 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 face the same problem: types must be written explicitly, at both the introduction and the elimination of polymorphism.

Recent results on the undecidability of type inference for system F [Wells, 1994, Kfoury and Wells, 1994, Pfenning, 1993] 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. We do not infer higher-order types and thus avoid higher-order unification, undecidable in general. Furthermore, we 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 if it is 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 already have known types. This information is only waiting to be used appropriately.

In comparison to previous works, we remove the requirement for type annotations at the elimination of polymorphism by using type inference to propagate explicit type information between different points of the program. In our proposal, tagging values of polymorphic types with type symbols becomes superfluous. 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, along with some syntactic comparisons. Section 4 shows how our system can be used to provide polymorphic methods for Objective ML, in an almost transparent way. In section 5 we discuss how the value-only restriction to polymorphism can be applied here. Lastly, we compare with related works, and conclude. Proofs of main theorems are given in appendix.

1 INFORMAL APPROACH

In this section we present our solution informally. We first introduce 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. We write $x \stackrel{\triangle}{=} a$ to introduce a meta-level name x for a formal expression a.

1.1 A naive solution

The self-application term $\mathit{self} \stackrel{\triangle}{=} \lambda f.\, f\, f$ cannot be typed in ML; however, we can easily type it in system F if we add proper type annotations. While this expression is not very interesting for itself, a few variations on it are sufficient to illustrate most aspects of type inference in the presence of higher-order types. Useful examples can be found in section 4 in addition to those suggested in the introduction.

The expression $\texttt{let}\ f = \mathit{id}\ \texttt{in}\ f\ f$ where $\mathit{id} \stackrel{\triangle}{=} \lambda x.\, x$ (the polymorphic identity function) is typable in ML. One can see let-definitions as a special syntax, combined with a special typing rule, for the application $(\lambda x.\, a_2)\ a_1$. Let us exercise by replacing the `let` polymorphic binding by first-class polymorphism. The identity id has type $\alpha \to \alpha$ where α can be universally quantified. We shall write $[\mathit{id} : \forall \alpha.\alpha \to \alpha]$ for the *creation* (or *introduction*) of the polymorphic value wrapping id with the polymorphic type $\forall \alpha.\alpha \to \alpha$. As usual in ML, we distinguish between first-class *simple types* (or *types* for short) and *polymorphic types*. Thus, we explicitly coerce the polymorphic type $\forall \alpha.\alpha \to \alpha$ to a simple type $[\forall \alpha.\alpha \to \alpha]$ using the type constructor $[\_]$ for that purpose. We call $[\forall \alpha.\alpha \to \alpha]$ a polytype, which is (a particular form of) a simple type.

Let id_1 be the expression $[\mathit{id} : \forall \alpha.\alpha \to \alpha]$, which has type $[\forall \alpha.\alpha \to \alpha]$. As any first-class value, id_1 can be passed to other functions, stored in data-structures, *etc.*

For instance $(id_1, 1)$ is a pair of type $([\forall\alpha.\alpha \to \alpha] \times \texttt{int})$. Such a wrapped 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, $\langle id_1 \rangle$ 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 $\alpha \to \alpha$, making its typing behavior just the same as the polymorphic identity function *id*.

The raw expression *self* 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 by a type constraint on the argument: $\lambda f\colon [\forall\alpha.\alpha \to \alpha].\, \langle f \rangle\ f$. The first occurrence of f in the body is opened to eliminate polymorphism before it is applied. The following definition expression is well-typed:

$$self_1 \stackrel{\triangle}{=} \lambda f\colon [\forall\alpha.\alpha \to \alpha].\, \langle f \rangle\ f \qquad : [\forall\alpha.\alpha \to \alpha] \to [\forall\alpha.\alpha \to \alpha]$$

So are the two following variants:

$$self_2 \stackrel{\triangle}{=} \lambda f\colon [\forall\alpha.\alpha \to \alpha].\, \langle f \rangle\ \langle f \rangle \qquad : [\forall\alpha.\alpha \to \alpha] \to \alpha' \to \alpha'$$
$$self_3 \stackrel{\triangle}{=} \lambda f\colon [\forall\alpha.\alpha \to \alpha].\, [\langle f \rangle\ \langle f \rangle : \forall\alpha.\alpha \to \alpha] \qquad : [\forall\alpha.\alpha \to \alpha] \to [\forall\alpha.\alpha \to \alpha]$$

In $self_2$, the occurrence of f in the argument position is also opened, so the result type is no longer a polytype. In $self_3$, polymorphism is lost as in $self_2$, then it is recovered explicitly. Finally, we can apply $self_1$ to the wrapped identity function id_1:

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

More interestingly, the following expression is also well-typed

$$(\lambda u.\, u\ id_1)\ self_1 \qquad : [\forall\alpha.\alpha \to \alpha]$$

There is no term typable in ML that has the same erasure (untyped λ-term) as this one. Note that no type annotation is needed on u; although u has a polytype as result, it is not opened locally.

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 well-typed? In order to avoid higher-order types, we accept to reject those examples. Our modest goal is to keep track of user-provided polymorphism, but never guess polymorphism from scratch.

On the other hand, forbidding lambda abstraction of an unspecified type to be a polytype is too restrictive. This would violate the assumption that polytypes are regular ML types, which can be substituted for any type variable. Thus, if $\lambda x.x$ has type $\alpha \to \alpha$, it should also have type $[\sigma] \to [\sigma]$ for any polymorphic type σ. Actually, for practical programming, it is important that $\lambda x. x$ possesses 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. The former expression is needed as soon as we are using polymorphic values inside generic data structures, such as lists, and use polymorphic functions to extract them. The later allows η-expansion, for instance to reorder the arguments of a function, without any superfluous type annotation.

When typing $\lambda f. \langle f \rangle\ f$, variable f is first given an unknown type τ. Guessing $[\forall \alpha. \alpha \to \alpha]$ for τ would be correct, but not principal, since $[\forall \alpha. \alpha \to \alpha \to \alpha]$ would also be a possible type for τ. Conversely, the expression $\lambda f. \texttt{if true then} \langle f \rangle\ f\ \texttt{else}\ (f : [\forall \alpha. \alpha \to \alpha])$ would have as only possible type $[\forall \alpha. \alpha \to \alpha] \to [\forall \alpha. \alpha \to \alpha]$, since there is an explicit annotation on f. However, we prefer to also reject this program. Informally, type inference would imply backtracking: f is first assumed of unknown type τ; we cannot type $\langle f \rangle$ so we backtrack; typing the annotation 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[1] and we would rather fail in every case where some inference order would fail.

Worse, typing constraints may disappear during reduction. Traditionally, this is not a problem since this only allows to infer better types. However, in our case, the removal of polytype constraints will leave some polytypes unspecified and lead to failure. Consequently, we would lose the subject reduction property. The expression $\lambda f. \texttt{if true then} \langle f \rangle\ f\ \texttt{else}\ (f : [\forall \alpha. \alpha \to \alpha])$ reduces to $\lambda f. \langle f \rangle\ f$ but the latter is not typable.

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 variable ε (label for short). That is, we write $[\sigma]^\varepsilon$ rather than $[\sigma]$.

To ensure that an expression was correctly annotated, in an elimination $\langle a \rangle$, the type of a must be of the form $\forall \varepsilon. [\sigma]^\varepsilon$. This prevents negative occurrences of the type annotation (such as in the context or on the left hand-side of an arrow), so proving that it must have been user-provided. Expressions of the form $\lambda f. \langle f \rangle\ f$ are not allowed; the type of f is a simple type, which includes $[\sigma]^\varepsilon$, but not $\forall \varepsilon. [\sigma]^\varepsilon$.

Annotations do introduce polymorphism. We may write $\lambda x. \texttt{let}\ f = (x : [\sigma]^\varepsilon)\ \texttt{in}\ \langle f \rangle\ f$, where the type annotation on x is a polytype. Such an annotation allows label variables to be renamed apart in the type of f and then abstracted over in generalizing the type of the let-definition, thus allowing f to be used polymorphically in the let body. For convenience, we write $\lambda x{:}\tau. a$ as an abbreviation for $\lambda x. \texttt{let}\ x = (x : \tau)\ \texttt{in}\ a$. Hence $\lambda f{:}[\forall \alpha. \alpha \to \alpha]^\varepsilon. \langle f \rangle\ f$ is well-typed.

[1]ML typability is exponentially hard in theory, but it is almost linear in practice; here, the combinatorial explosion would likely make type inference exponential in practice.

Annotations must be correctly introduced. The expression $\lambda f.$`if true then` $\langle f \rangle\ f$ `else` $(f : [\forall\alpha.\alpha \to \alpha]^{\varepsilon})$ fails to type. The type $[\sigma]^{\varepsilon}$ of the else-branch is transmitted to the then-branch by unification. However, it is also simultaneously transmitted to the binding occurrence; hence, the label variable ε also appears in the type context and cannot be generalized; therefore $\langle f \rangle$ is ill-typed. An explicit type annotation is required on the then-branch: $\langle f : [\sigma]^{\varepsilon'} \rangle\ f$. This has the effect of renaming ε into a fresh label variable ε' that does not occur in the context so that it can be generalized. For convenience, we write $[\sigma]$ instead of $[\sigma]^{\varepsilon}$ when the label ε is anonymous, *i.e.* when it does not appear anywhere else in the program, such as ε' in the above example.

Another subtle point is where to bind type variables that occur free in a type annotation $(a : \tau)$. Traditionally, these are shared between several type annotations, and thus implicitly bound at a higher level according to scoping rules that depend on the ML dialect. In our system, we chose to bind them (existentially) in the type constraint where they occur. That is, they are never shared between two different type annotations. This is simpler than defining specific scoping rules; and sharing a type variable between several annotations could lead to lose polymorphism unexpectedly.

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 $\varepsilon \in \mathcal{E}$. The syntax of types is:

$\tau ::= \alpha \mid \tau \to \tau \mid [\sigma]^{\varepsilon}$	(Simple) types
$\sigma ::= \tau \mid \forall\alpha.\sigma$	Polymorphic types
$\varsigma ::= \sigma \mid \forall\varepsilon.\varsigma$	Type schemes
$\xi ::= \alpha \mid \varepsilon$	Variables

The construct $[\sigma]^{\varepsilon}$ is used to coerce a polymorphic type σ to a type. We call $[\sigma]^{\varepsilon}$ a weak polytype. The label variable ε is used to keep track of sharing between weak polytypes. When an expression has a polytype $[\sigma]^{\varepsilon}$ and the label variable ε can be generalized, then the polytype can be eliminated and the expression can be given the polymorphic type σ. We do not allow polymorphic labels in polymorphic types σ, since this would not add any power to the system (it would be redundant with explicit type annotations —see section 2.5).

Free type variables and free labels of a type scheme (which may be a simple type) ς are written $FV(\varsigma)$ and $FL(\varsigma)$ respectively and are defined as usual. In a type scheme $\forall\xi.\varsigma$, $\forall$ acts as a quantifier, and the variable ξ is bound (*i.e.* not free) in $\forall\xi.\varsigma$. We consider type schemes equal by renaming and reordering of bound variables and labels, and removal of useless quantifiers (*i.e.* $\forall\xi.\tau \equiv \tau$ whenever variable ξ

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

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

$$\frac{A \vdash a : \forall \varepsilon.\varsigma}{A \vdash a : \varsigma\{\varepsilon'/\varepsilon\}}\ (\textsc{Inst-E}) \qquad \frac{A \vdash a_1 : \varsigma \qquad A, x : \varsigma \vdash a_2 : \tau}{A \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau}\ (\textsc{Let}) \qquad \frac{A \vdash a : \tau_1 \qquad (\tau_1 : \tau : \tau_2)}{A \vdash (a : \tau) : \tau_2}\ (\textsc{Ann})$$

$$\frac{A \vdash a : \sigma_1 \qquad (\sigma_1 : \sigma : \sigma_2)}{A \vdash [a : \sigma] : [\sigma_2]^{\varepsilon}}\ (\textsc{Intro}) \qquad \frac{A \vdash a : \forall \varepsilon.[\sigma]^{\varepsilon}}{A \vdash \langle a \rangle : \sigma}\ (\textsc{Elim})$$

Figure 1: Typing rules

is not free in τ). We write $\{\tau_1, \dots \tau_n/\alpha_1, \dots \alpha_n\}$ for the simultaneous substitution of variables $\alpha_1, \dots, \alpha_n$ by $\tau_1, \dots, \tau_n$, respectively. As usual, bound variables and bound labels are renamed by substitutions so that free variables of τ_i's can remain unchanged without being captured. For example $(\alpha \to [\forall \beta.\beta \to \alpha]^{\varepsilon})\{\tau/\alpha\}$ is $\tau \to [\forall \beta.\beta \to \tau]^{\varepsilon}$ provided β is not free in τ. An instance of a type scheme $\forall \bar{\varepsilon}, \bar{\alpha}.\tau_0$ is $\tau\{\bar{\varepsilon}', \bar{\tau}/\bar{\varepsilon}, \bar{\alpha}\}$. A generic instance of a type scheme ς is a type scheme $\forall \bar{\xi}.\tau$ such that τ is an instance of ς and bound variables $\bar{\xi}$ do not occur free in ς.

Expressions are those of core ML (left) plus three new constructs (right): introduction and elimination of first-class polymorphism and type annotation.

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

Typing rules are given in figure 1. Typing judgments are of the form $A \vdash a : \varsigma$ where A is a set of typing assumptions binding expression variables to type schemes. The extension of a set of typing assumptions A with a new binding $x : \varsigma$ is written $A, x : \varsigma$; it overrides any previous binding of x in A. All typing rules but the last three ones are standard. Rules Ann and Intro use an auxiliary relation $(\_ : \_ : \_)$. Given a polymorphic type σ, we write $(\sigma_1 : \sigma : \sigma_2)$ if there exists a substitution θ from type variables to simple 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, $(\rho_1(\sigma) : \sigma : \rho_2(\sigma))$ for any label renamings ρ_1 and ρ_2. 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. The use of θ implements the local quantification of user-given type variables that we stated in the informal presentation. 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. Note that σ is user-given and the substitution θ is not applied to σ.

This relation is used to type explicit annotations. For typechecking purposes, 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 scheme $\forall \bar{\varepsilon}_1, \bar{\varepsilon}_2, FV(\tau).\ \tau\{\bar{\varepsilon}_1/\bar{\varepsilon}\} \to \tau\{\bar{\varepsilon}_2/\bar{\varepsilon}\}$ where $\bar{\varepsilon}_1$ and $\bar{\varepsilon}_2$ are different renamings of the tuple $\bar{\varepsilon}$ of all label variables of τ. That is, to type an expression $(a : \tau)$, let τ_1 and τ_2 be two copies of τ where their labels have been renamed, and θ be a substitution such that a has type $\theta(\tau_1)$; then $(a : \tau)$ has type $\theta(\tau_2)$. We kept annotation as a primitive construct because the dynamic semantics is simpler to define this way, but this is mainly a matter of exposition.

Rule INTRO uses the same relation, except that polymorphic types replace simple types. To type $[a : \sigma]$, let σ_1 and σ_2 be two copies of σ where labels have been renamed; find a substitution θ such that a has type $\theta(\sigma_1)$ (*i.e.* $\theta(\sigma_1)$ is a generic instance of the principal type scheme of a); then $[a : \sigma]$ has type $[\theta(\sigma_2)]^{\varepsilon}$ for any label ε.

Last, rule ELIM 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 $f : [\sigma]^{\varepsilon_1}$:

$$
\begin{array}{r}
(\sigma \equiv \forall\alpha.\alpha\to\alpha)\\
(\alpha \leftarrow [\sigma]^{\varepsilon_1})
\end{array}
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\textsc{Var})\;\dfrac{}{A \vdash f : [\sigma]^{\varepsilon_1}} \qquad ([\sigma]^{\varepsilon_1} : [\sigma]^{\varepsilon} : [\sigma]^{\varepsilon_2})}{A \vdash (f : [\sigma]^{\varepsilon}) : [\sigma]^{\varepsilon_2}}\;(\textsc{Ann})}{A \vdash (f : [\sigma]^{\varepsilon}) : \forall\varepsilon_2.[\sigma]^{\varepsilon_2}}\;(\textsc{Gen-E})}{A \vdash \langle f : [\sigma]^{\varepsilon}\rangle : \forall\alpha.\alpha\to\alpha}\;(\textsc{Elim})}{A \vdash \langle f : [\sigma]^{\varepsilon}\rangle : [\sigma]^{\varepsilon_1} \to [\sigma]^{\varepsilon_1}}\;(\textsc{Inst-V})}{}
$$

$$
\dfrac{\dfrac{\vdots \qquad \dfrac{}{A \vdash f : [\sigma]^{\varepsilon_1}}\;(\textsc{Var})}{A \vdash \langle f : [\sigma]^{\varepsilon}\rangle\ f : [\sigma]^{\varepsilon_1}}\;(\textsc{App})}{\vdash \lambda f.\,\langle f : [\sigma]^{\varepsilon}\rangle\ f : [\sigma]^{\varepsilon_1} \to [\sigma]^{\varepsilon_1}}\;(\textsc{Fun})
$$

2.2 *Dynamic semantics*

We give a reduction semantics for the core language. Actually we define two semantics: a free reduction semantics for which we prove only subject reduction, and a call-by-value semantics for which we prove full type soundness.

A one step reduction is either an immediate reduction (the label on the reduction is indicative and optional):

$$
\begin{array}{rcl}
(\lambda x.\,a)\ b & \xrightarrow{\mathsf{Fun}} & a\{b/x\} \\
\mathtt{let}\ x = b\ \mathtt{in}\ a & \xrightarrow{\mathsf{Let}} & a\{b/x\} \\
\langle [a : \forall\bar{\alpha}.\tau] \rangle & \xrightarrow{\mathsf{Elim}} & (a : \tau) \\
(a : \tau_2 \to \tau_1)\ b & \xrightarrow{\mathsf{Tfun}} & (a\ (b : \tau_2) : \tau_1) \\
([a : \forall\bar{\alpha}.\tau] : [\sigma]^{\varepsilon}) & \xrightarrow{\mathsf{Tint}} & [(a : \tau) : \sigma] \\
(a : \alpha) & \xrightarrow{\mathsf{Tvar}} & a
\end{array}
$$

or obtained by induction (E is any term context with a single hole):

$$\frac{a_1 \longrightarrow a_2}{E\{a_1\} \longrightarrow E\{a_2\}}$$

Note that the meta variable α, in rule TVAR, stands for a type variable and not for an arbitrary type. 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 effectively reduced by rules FUN, LET, and ELIM, we need the rules TFUN and TINT to maintain this type information. These rules are needed for the following example: $\lambda f.\langle (\lambda x.x : [\sigma] \to \alpha)\ f \rangle$ reduces to $\lambda f.\langle f : [\sigma] \rangle$ which would not be typable but for the annotation. Rule TVAR erases vacuous 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 $\stackrel{\sim}{\longrightarrow}$ and a type-erasure $\sim$ so that if $a_1 \longrightarrow a_2$, then $\widetilde{a}_1 \stackrel{\sim}{\longrightarrow} \widetilde{a}_2$ or $\widetilde{a}_1$ and $\widetilde{a}_2$ are equal.

We now define the call-by-value operational semantics by restricting the free reduction semantics. Evaluation contexts (used for the above induction rule) are then

$$E ::= \{\} \mid E\ a \mid v\ E \mid \mathtt{let}\ x = E\ \mathtt{in}\ a \mid [E : \sigma] \mid (E : \tau) \mid \langle E \rangle$$

and the strategy is fixed so that inner redexes are reduced first. This is implemented by substituting a value meta variable v for term meta variables a or b when they appear at evaluable positions in the reduction rules. Values v are defined as follows:

$$
\begin{array}{l}
v ::= w \mid [v : \sigma] \\
w ::= \lambda x.\,a \mid (w : \tau_1 \to \tau_2)
\end{array}
$$

By default, reduction will always refer to free reduction.

2.3 Type soundness

We could easily show that evaluation cannot go wrong by means of translation into system F. We prefer to prove it in a more direct way. Subject reduction is an intermediate result of the 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 the introduction of polytypes; 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. We chose a logic rather than a syntax directed presentation of typings rules, since this is here much more concise. We can still recover the benefits of a syntax directed presentation by using canonical derivations. 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 INTRO, ELIM, the left premise of rule LET, or another rule GEN.
- rule INST may only occur right after rule VAR, rule ELIM, or another rule INST.

Canonical derivations have been defined to validate the following lemma.

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

Another classical key 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 (see the last paragraph of section 1.3).

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.*

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

This simplifies the statement of subject reduction, expressed for free reduction.

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 prevent from type errors. It must be complemented by the following result, which we only express for call-by-value semantics.

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

Type soundness of the call-by-value semantics is a straightforward combination of the two previous theorems.

2.4 *Type inference*

We present both unification and type inference as constraint solving using rewriting techniques. This formalism, now well-established [Jouannaud and Kirchner, 1991], has several significant advantages over older, more algorithmic presentations of unification algorithms: renaming and introduction of fresh variables is rigorously and simply formalized by existential binders; sharing, hence recursive types, is formally dealt with by the use of multi-equations instead of simple equations[2]; the presentation with rewriting constraints is also more modular, which eases proofs as well as further extensions. The same framework can also be used for type inference, treating type inference problems as unification problems [Rémy, 1992]. Indeed, solutions of type inference problems are also sets of substitutions. All the previous benefits of treating unification as constraint solving also apply to type inference. In particular, type inference can be specified and proved correct independently of any strategy. A top-down, bottom-up, or any other —even non-deterministic— terminating strategy can be chosen later, or remain unspecified.

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. Reusing the framework of constraint solving, we show that the addition of first-class polymorphism retain the flexibility and modularity of type inference. Simultaneously, we provide formal, general, and efficient unification and type inference algorithms (no use of "fresh variables", preservation of sharing, treatment of recursive types[3]).

More precisely, the formalism used is that of conditional rewriting. For clarity of presentation, we distinguish between two kinds of conditions. Those that can always be satisfied are written **let** *condition* **in** *rule*; they amount to a convenient notation for pattern matching. Other conditions may fail, providing dynamic control during the inference process; they are written **if** *condition* **then** *rule*.

Unification for simple types First, we remind unification for simple types. In this part only, we exclude polytypes from simple types, still ranged over by letter τ. A *unification problem*, also called a *unificand*, is a formula U defined by the following grammar.

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

The symbols $\top$ and $\bot$ are respectively the trivial and unsatisfiable unification problems. We treat them as a unit and a zero for $\wedge$. That is $U \wedge \top$ and $U \wedge \bot$ are equal to U and $\bot$, respectively. We also identify $\top$ with singleton multi-equations. That

[2] Multi-equations can be easily mapped to equivalence classes as in [Huet, 1976], as well as to the mutable structures that are used in destructive unification algorithms.

[3] Recursive types are correctly handled by our algorithms, although they are not considered in the proofs.

is, we can always consider that a unification problem U contains at least one multi-equation α or $\alpha \doteq e$ for each variable α 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 $\exists$ will be needed later for polytypes. It 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. This makes multi-equations behave as multi-sets of types.

The substitution of types is extended to unificands in a straightforward way. For existentials, the application of a substitution θ to a unificand $\exists\alpha.U$ is the unificand $\exists\alpha'.\theta(U\{\alpha'/\alpha\})$ where α' is chosen outside of both the domain and the codomain of θ and outside free variables of U.

A substitution θ is a solution of a multi-equation if it sends all types of the multi-equation to the same codomain. 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 $U\{\alpha'/\alpha\}$ where α' is chosen outside of both the domain and the codomain of θ and outside free variables of U.

Two unification problems are equivalent if they have the same set of solutions. All previous structural equalities are indeed equivalences. We write $U_1 \equiv U_2$ when the unification problems U_1 and U_2 are equivalent. We also write $U_1 \Rrightarrow U_2$ to mean that the unification problem U_1 can be rewritten into the equivalent unification problem U_2. Finally, a solution θ is a principal solution of a unification problem U if any other solution can be obtained by (left) composition with the substitution θ.

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 type that contains α' as a free variable. A unification problem is strict if $\prec_U$ is irreflexive. Note that strictness is syntactic and is not preserved by equivalence. Intuitively, strictness corresponds to the absence of immediate cycles. However, it does not detect potential cycles that may appear after some computation steps. Still, for fully merged and decomposed unification problems, *i.e.* when the rules MERGE and DECOMPOSE cannot be applied anymore, strictness is equivalent to the fact that if there is a solution then there is a finite solution.

A problem is in solved form if it is either $\perp$ or $\top$, or if it is strict, merged, decomposed, and of the form $\exists\bar{\alpha}.\ \bigwedge_{i\in 1..n} e_i$. In particular, each multi-equation e_i contains at most one non-variable type, and if $i \neq j$ then e_i and e_j contain no variable type in common. An explicit principal solution θ can be read straightforwardly from a problem in solved form. We also 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 $\perp$. This is consistent with the previous notation since θ could be seen as $\wedge_{\alpha\in dom(\theta)}\alpha \doteq \theta(\alpha)$ whenever its domain and codomain are disjoint.

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

OCCUR-CHECK
if $\prec_U$ is not strict **then**
$U \Rrightarrow \bot$
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
if $size(\tau_1 \to \tau_2) \leq size(\tau_1' \to \tau_2')$ **then**
$\tau_1 \to \tau_2 \doteq \tau_1' \to \tau_2' \doteq e \Rrightarrow \tau_1 \to \tau_2 \doteq e \wedge \tau_1 \doteq \tau_1' \wedge \tau_2 \doteq \tau_2'$

Figure 2: First-order unification for simple types

formulas by rewriting any sub-formula. We write $size(\sigma)$ the size of polymorphic type σ counted as the number of occurrences of symbols $(\_ \to \_)$ or $[\_]$ in σ. These rules are all standard. 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 for simple-types with polytypes We now extend types to polytypes $[\sigma]^\varepsilon$. Consistently, we extend type variables with label variables. Hence, substitutions are from type variables to polytypes and from label variables to label variables, unless otherwise specified. In order to allow a natural decomposition of polytypes, we extend typing problems with equations between polymorphic types.

$$U ::= \ldots \mid \sigma \stackrel{\forall}{=} \sigma$$

These are not multi-equations. In particular, a variable cannot be equated to an arbitrary polymorphic type. For instance, $\alpha' \stackrel{\forall}{=} \forall\alpha.\alpha \to \alpha$ does not have any solution. Thus, equations involving polymorphic types are never merged.

A substitution θ is a solution of a polytype equation $\forall\bar{\alpha}.\tau \stackrel{\forall}{=} \forall\bar{\alpha}'.\tau'$ if $\theta(\forall\bar{\alpha}.\tau) = \theta(\forall\bar{\alpha}'.\tau')$, where equality is the usual equality for polymorphic types in ML, *i.e.* it is taken modulo reordering and renaming of universal quantifiers and removal of useless universal variables. This is equivalent to the existence of

- two injective substitutions ρ and ρ' of respective domains $\bar{\alpha}$ and $\bar{\alpha}'$ and of codomain $\bar{\alpha}\bar{\alpha}'$ and
- a renaming η from $\bar{\alpha}\bar{\alpha}'$ outside of free variables of θ, τ, τ', and $\bar{\alpha}\bar{\alpha}'$

such that $\theta \circ \eta$ is a solution of $\rho(\tau) = \rho'(\tau')$. We could solve such unification problems by first unifying $\rho(\tau)$ and $\rho'(\tau')$ and then checking the constraints. However,

Decompose-Poly
  if $size(\sigma) \leq size(\sigma')$ **then**
  $[\sigma]^{\varepsilon} \doteq [\sigma']^{\varepsilon'} \doteq e \Rrightarrow [\sigma]^{\varepsilon} \doteq e \wedge \varepsilon \doteq \varepsilon' \wedge \sigma \stackrel{\forall}{=} \sigma'$
Clash
  $[\sigma]^{\varepsilon} \doteq \tau \to \tau' \doteq e' \Rrightarrow \bot$
Polytypes
  let $\bar{\alpha} \cap \bar{\alpha}' = \emptyset$ **and** $\bar{\alpha} \cap FV(\tau') = \emptyset$ **and** $\bar{\alpha}' \cap FV(\tau) = \emptyset$ **in**
  $\forall \bar{\alpha}.\tau \stackrel{\forall}{=} \forall \bar{\alpha}'.\tau' \Rrightarrow \exists\, \bar{\alpha}\bar{\alpha}'.\, \tau \doteq \tau' \wedge \bar{\alpha} \leftrightarrow \bar{\alpha}'$
Renaming-True
  let $\bar{\alpha} = (\alpha_i)^{i \in 1..n+p}$ **and** $\bar{\alpha}' = (\alpha'_i)^{i \in 1..n+q}$ **in**
  $\exists\, \bar{\alpha}\bar{\alpha}'.\, (\alpha_i \doteq \alpha'_i)^{i \in 1..n} \wedge \bar{\alpha} \leftrightarrow \bar{\alpha}' \Rrightarrow \top$
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** $\gamma \doteq \tau \doteq e \wedge \bar{\alpha} \leftrightarrow \bar{\alpha}' \Rrightarrow \bot$

Figure 3: First-order unification for simple types with polytypes

this would force some unnecessary dependence. Note that η is only here for technical purposes, and can be omitted if θ is disjoint from $\bar{\alpha}\bar{\alpha}'$. This can be dealt with by existential quantification of unificands.

Without loss of generality, we can restrict ourselves to the case where $\bar{\alpha} \cap \bar{\alpha}'$, $FV(\tau) \cap \bar{\alpha}'$, and $FV(\tau') \cap \bar{\alpha}$ are all empty sets. We refer to these hypotheses by condition (H). We write the sum of two substitutions of disjoint domains $\theta + \theta'$ that maps variables of $dom(\theta)$ and $dom(\theta')$ to their image by θ or θ', respectively. We write $\theta \upharpoonright W$ for the restriction of the substitution θ to the set of variables W, that is, the substitution equal to θ on $dom(\theta) \cap W$ and to the identity everywhere else. We write $V \setminus W$ for the set difference between V and W, *i.e.* the set of all elements that are in V but not in W. Consistently, we write $\theta \setminus W$ for the restriction of a substitution outside of a set of variables W, that is the restriction of θ to $dom(\theta) \setminus W$, or formally, $\theta \upharpoonright (dom(\theta) \setminus W)$.

Let θ' be $(\eta + \eta^{-1}) \circ \theta \circ \eta \circ (\rho + \rho')$, which decomposes as $(\eta \circ \theta \setminus \bar{\alpha}\bar{\alpha}') + (\rho + \rho')$. (If θ is disjoint from $\bar{\alpha}\bar{\alpha}'$, then θ' is simply $\theta \circ (\rho + \rho')$, which decomposes into $\theta + \rho + \rho'$.) The substitution θ' satisfies the three following properties:

(1) $\theta'(\tau) = \theta'(\tau')$,

(2) $\theta' \upharpoonright \bar{\alpha}$ and $\theta' \upharpoonright \bar{\alpha}'$ are injective in $\bar{\alpha}\bar{\alpha}'$, and

(3) no variable of $\bar{\alpha}\bar{\alpha}'$ appears in $codom(\theta' \setminus \bar{\alpha}\bar{\alpha}')$.

Conversely, a substitution θ' satisfying these three conditions is a solution of $\forall \bar{\alpha}.\tau \stackrel{\forall}{=} \forall \alpha'.\tau'$.

The condition (1) above is a unification problem. We introduce a new kind of unificands $\bar{\alpha} \leftrightarrow \bar{\alpha}'$ whose solutions are substitutions satisfying the conditions (2) and (3) simultaneously. We consider $\bar{\alpha}$ and $\bar{\alpha}'$ as multi-sets (*i.e.* the comma is associative and commutative). In order to avoid special cases, we also 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 $\stackrel{\forall}{=}$ and $\leftrightarrow$ are commutative. Then θ is a solution of $\forall\bar{\alpha}.\tau \stackrel{\forall}{=} \forall\bar{\alpha}'.\tau'$ under the assumption (H), if and only if it is a solution $\exists\,\bar{\alpha}\bar{\alpha}'.\,(\tau \doteq \tau' \wedge \bar{\alpha} \leftrightarrow \bar{\alpha}')$. Note that unificands are no longer stable by arbitrary substitutions as long as they contain free variables appearing in renaming unificands (otherwise, renaming unificands could even become ill-formed.) Still, unificands remain stable by renamings. Indeed this is necessary to give meaning to existentially quantified unificands.

Rules for unification with polytypes are those of figures 2 and 3 together. Rule CLASH handles type incompatibilities. Rule POLYTYPES transforms polytype equations as described above. Rule RENAMING-TRUE allows to remove a satisfiable renaming constraint that became garbage, *i.e.* independent of all other multi-equations. 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 γ 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 rule RENAMING-FALSE applies. Therefore, renaming constraints can always be eliminated.

Theorem 3 *Given a unification problem U, there exists a most general unifier* θ *which is computed by the set of rules in figures 2 and 3, or there is no unifier and U reduces to $\bot$.*

Type inference For type inference, we extend unificands 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 substitution. Thus, typing problems can be treated as unification problems, following [Rémy, 1992]. The rules for solving typing problems are given in figure 4. The generalization $Gen\,(\sigma, A)$ 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. To lighten the presentation, we leave it implicit that whenever we write $\exists\,\bar{\xi}.\,\theta$, variables $\bar{\xi}$ are assumed to be distinct from all other variables appearing in the rule.

The rewriting for type inference closely follows typing rules given in 1, except that we are assuming a syntactic presentation enforcing canonical derivations where rules VAR and ELIM are combined with (followed by) rule INST- and rules LET and USE are combined with (preceded by) rule GEN as in canonical forms. Rules VAR, FUN, APP, LET are the same as for ML [Rémy, 1992]. The remaining rules are new but unsurprising. Their close correspondence with rules of figure 1 is made in the proof of soundness and completeness of type inference given in appendix A.

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

Figure 4: Rewriting rules for type inference.

Theorem 4 *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, or there is no solution and the rules reduce to* $\bot$.

2.5 Polymorphic labels in polytypes

We did not allow labels in polymorphic types σ. We show here that this would not increase expressiveness. In this section, we consider an alternative type system, called the *extended* type system, where extended polytypes are of the form $[\varsigma]^{\varepsilon}$ instead of $[\sigma]^{\varepsilon}$. Typing rules are unchanged.

To show that this does not increase expressiveness, we define a translation $\langle\!\langle \_ \rangle\!\rangle_E$ from extended type schemes to type schemes. The translation is parameterized by a set of label variables E that is omitted when empty. For simplicity, we suppose that all quantified labels have different names in the definition of the translation:

$$
\begin{array}{rcll}
\langle\!\langle \alpha \rangle\!\rangle_E & \stackrel{\triangle}{=} & \alpha & \\
\langle\!\langle \tau_1 \to \tau_2 \rangle\!\rangle_E & \stackrel{\triangle}{=} & \forall \bar{\varepsilon}_1 \bar{\varepsilon}_2 . \tau'_1 \to \tau'_2 & \text{if } \langle\!\langle \tau_i \rangle\!\rangle_E = \forall \bar{\varepsilon}_i . \tau'_i \\
\langle\!\langle [\forall \bar{\varepsilon} . \sigma]^{\varepsilon_0} \rangle\!\rangle_E & \stackrel{\triangle}{=} & \forall \bar{\varepsilon} \bar{\varepsilon}' . [\sigma']^{\varepsilon_0} & \text{if } \varepsilon_0 \in E \text{ and } \langle\!\langle \sigma \rangle\!\rangle_{\bar{\varepsilon} \cup E} = \forall \bar{\varepsilon}' . \sigma' \\
\langle\!\langle [\forall \bar{\varepsilon} . \sigma]^{\varepsilon_0} \rangle\!\rangle_E & \stackrel{\triangle}{=} & \forall \bar{\varepsilon}' . [\sigma']^{\varepsilon_0} & \text{if } \varepsilon_0 \notin E \text{ and } \langle\!\langle \sigma \rangle\!\rangle_E = \forall \bar{\varepsilon}' . \sigma' \\
\langle\!\langle \forall \alpha . \sigma \rangle\!\rangle_E & \stackrel{\triangle}{=} & \forall \bar{\varepsilon} . \forall \alpha . \sigma' & \text{if } \langle\!\langle \sigma \rangle\!\rangle_E = \forall \bar{\varepsilon} . \sigma' \\
\langle\!\langle \forall \bar{\varepsilon} . \sigma \rangle\!\rangle & \stackrel{\triangle}{=} & \forall \bar{\varepsilon} . \langle\!\langle \sigma \rangle\!\rangle_{\bar{\varepsilon}} &
\end{array}
$$

Intuitively, the translation moves label quantifiers to the outer level. During this process however, label quantifiers that appear in a polytype whose label is itself not quantified are simply dropped. The translation is extended homomorphically to expressions and typing environments. Then, considering the judgment $A \vdash a : \varsigma$ in the extended type system, it is translated into the judgment $\langle\!\langle A \rangle\!\rangle \vdash \langle\!\langle a \rangle\!\rangle : \langle\!\langle \varsigma \rangle\!\rangle$ of our system. The latter has smaller type annotations since all label quantifiers are dropped in $\langle\!\langle a \rangle\!\rangle$. It is then easy to prove (by induction on the size of the former derivation) that whenever the former judgment is valid, so is the later.

2.6 Printing labels as sharing constraints

In this section, we propose an alternative interface to the user aimed at enhancing readability of types. It is also robust. However, it is slightly harder to present, formally. Hence, we followed the other, more traditional approach above for simplicity of presentation.

Labels are used to trace the sharing of polytypes. Types can be restricted so that two polytypes with the same label are necessarily equal. This property is not required in the present type system, but it is stable: if satisfied by all initial type assumptions in A and type annotations in a, then it remains valid in all types appearing in a principal derivation of $A \vdash a : \tau$. The grammar of types can be extended with a sharing construct[4]:

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

Using sharing, 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 types

[4]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.

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 present in the solved form. Actually, only sharing involving polytypes needs to be printed; all other sharing can be ignored.

For instance, the expression $\lambda x.\,(x : [\sigma])$ has type $[\sigma] \to [\sigma]$, which would have previously been written $\forall \varepsilon, \varepsilon'.[\sigma]^{\varepsilon} \to [\sigma]^{\varepsilon'}$. Conversely, the expression $\lambda x.\,\mathtt{let}\; y = (x : [\sigma]^{\varepsilon})\;\mathtt{in}\; x$ has type $(\alpha \to \alpha\;\mathtt{where}\;\alpha = [\sigma]^{\varepsilon})$ which would have been written $\forall \varepsilon.[\sigma]^{\varepsilon} \to [\sigma]^{\varepsilon}$, where the two polytypes share the same label.

Both notations (sharing constraints and label variables) actually coincide when all polytypes are anonymous (*i.e.* no label variable occur twice) and polytypes are simply written $[\sigma]$. For instance, $\lambda x.\,(x : [\sigma])$ has type $[\sigma] \to [\sigma]$. This is an important case, since the only types the user actually needs to write are of this form. Indeed, types written by the user are only type annotations, which become more general by removing sharing constraints. More precisely, if σ' is a polymorphic type obtained from σ by a label substitution ρ, then for any expression a, we have $(a : [\sigma]) \subset (a : [\sigma'])$ and $[a : \sigma] \subset [a : \sigma']$. This is an easy consequence of σ being more general than σ'.

Thus, the user never needs to write labels or sharing constraints, but he must read them in both inferred types and type-error messages.

3 ENCODINGS

In this section, we give encodings in our language for both explicit polymorphism with datatypes and system F. This last encoding is direct, and makes our language an alternative to system F. We also compare the use of explicit type information between system F and our proposal.

3.1 Type annotation on arguments

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

$$\text{(POLY-FUN)}\quad \frac{A, (x : \forall \bar{\varepsilon}.\tau_2) \vdash a : \tau' \qquad (\tau_1 : \tau : \tau_2) \qquad \bar{\varepsilon} \cap FL(\tau_1) = \emptyset}{A \vdash \lambda x{:}\tau.\,a : \tau_1 \to \tau'}$$

The derived reduction is $(\lambda x{:}\tau.\,a)\; b \xrightarrow{\mathsf{Fun}} a\{(b : \tau)/x\}$. Note that τ_1 and τ_2 are not just the results of renaming label variables of τ. They may also be an instance of τ. Hence, the set ε of generalized labels contains only labels corresponding to copies of those of τ and do not include any label that would have been brought by the instance of a free type variable of τ (since those would also appear in τ_1).

3.2 Polymorphic datatypes

Previous works have used data types to provide ML with explicit polymorphism [Läufer and Odersky, 1994, Rémy, 1994, Odersky and Läufer, 1996]. Omitting other aspects that are irrelevant here, all these works amount to an extension of ML with expressions of the form:

$$\begin{array}{rll} t ::= & \alpha \mid t \to t \mid T\ \bar{\alpha} & \text{Types} \\ s ::= & t \mid \forall \alpha . s & \text{Polymorphic types} \\ M ::= & x \mid M\ M \mid \lambda x . M \mid T\ M \mid T^{-1}\ M & \text{Terms} \\ & \mid \mathtt{type}\ T\ \bar{\alpha} = s\ \mathtt{in}\ M & \text{Type declarations} \end{array}$$

where T ranges over datatype symbols. In expressions, T and T^{-1} act as mutually inverse introduction and elimination functions to coerce the polymorphic type s into the simple type $T\ \bar{\alpha}$.

The translation is an inductive definition $\langle\!\langle \_ \rangle\!\rangle_\rho$. The environment ρ is a list of type definitions $\mathtt{type}\ T\ \bar{\alpha} = \sigma_0$ and $\rho(T)$ is the function $\Lambda\bar{\alpha}.\sigma_0$, *i.e.* given type arguments $\bar{\tau}$, it returns the type $\sigma_0\{\bar{\tau}/\bar{\alpha}\}$, using the right most definition of T in ρ. The translation of these types into types of our language is straightforward. The translation does not actually use type annotations smartly, and uses a single label ε. (While the program uses only one label, the type derivation need at least two other labels to type the elimination patterns $\langle\langle\!\langle M \rangle\!\rangle_\rho : [\rho(T)\ \bar{\alpha}]^\varepsilon\rangle$, locally.) It could also make all labels of the translation different, *i.e.* anonymous, but this is not needed.

$$\langle\!\langle \alpha \rangle\!\rangle_\rho \triangleq \alpha \qquad \langle\!\langle t_1 \to t_2 \rangle\!\rangle_\rho \triangleq \langle\!\langle t_1 \rangle\!\rangle_\rho \to \langle\!\langle t_2 \rangle\!\rangle_\rho \qquad \langle\!\langle T\ \bar{t} \rangle\!\rangle_\rho \triangleq [\langle\!\langle \rho(T)\ \bar{t} \rangle\!\rangle_\rho]^\varepsilon$$

$$\langle\!\langle \forall \alpha . s \rangle\!\rangle_\rho \triangleq \forall \alpha . \langle\!\langle s \rangle\!\rangle_\rho$$

We translate programs as follows.

$$\begin{array}{rcl@{\qquad}rcl} \langle\!\langle x \rangle\!\rangle_\rho & \triangleq & x & \langle\!\langle \mathtt{type}\ T\ \bar{\alpha} = t\ \mathtt{in}\ M \rangle\!\rangle_\rho & \triangleq & \langle\!\langle M \rangle\!\rangle_{\rho, \mathtt{type}\ T\ \bar{\alpha} = t} \\ \langle\!\langle \lambda x . M \rangle\!\rangle_\rho & \triangleq & \lambda x . \langle\!\langle M \rangle\!\rangle_\rho & \langle\!\langle M_1\ M_2 \rangle\!\rangle_\rho & \triangleq & \langle\!\langle M_1 \rangle\!\rangle_\rho\ \langle\!\langle M_2 \rangle\!\rangle_\rho \\ \langle\!\langle T\ M \rangle\!\rangle_\rho & \triangleq & [\langle\!\langle M \rangle\!\rangle_\rho : \rho(T)\ \bar{\alpha}] & \langle\!\langle T^{-1}\ M \rangle\!\rangle_\rho & \triangleq & \langle(\langle\!\langle M \rangle\!\rangle_\rho : [\rho(T)\ \bar{\alpha}]^\varepsilon)\rangle \end{array}$$

Indeed, the pattern $\langle(\_ : [\sigma])\rangle$ amounts to the explicit elimination of polymorphism (the explicit polytype annotation $(\_ : [\sigma])$ ensures that the polytype is anonymous.)

Since the elimination of polymorphism is always explicit in the translated terms, it can easily be shown that the translation of a well-typed term is well-typed. More precisely, we extend the translation $\langle\!\langle \_ \rangle\!\rangle_\_$ to typing environments as follows.

$$\langle\!\langle \emptyset \rangle\!\rangle_\rho \triangleq \emptyset \qquad \langle\!\langle A, x : \tau \rangle\!\rangle_\rho \triangleq \langle\!\langle A \rangle\!\rangle_\rho, x : \langle\!\langle \tau \rangle\!\rangle_{\rho, \lceil A \rceil} \qquad \langle\!\langle A, \mathtt{type}\ T\ \bar{\alpha} = s \rangle\!\rangle_\rho \triangleq \langle\!\langle A \rangle\!\rangle_\rho$$

$$\lceil \emptyset \rceil \triangleq \emptyset \qquad \lceil A, x : \tau \rceil \triangleq \lceil A \rceil \qquad \lceil A, \mathtt{type}\ T\ \bar{\alpha} = s \rceil \triangleq (\lceil A \rceil, \mathtt{type}\ T\ \bar{\alpha} = s)$$

For any term M, if $A \vdash M : t$, then $\langle\!\langle A \rangle\!\rangle \vdash \langle\!\langle M \rangle\!\rangle_{\lceil A \rceil} : \langle\!\langle t \rangle\!\rangle_{\lceil A \rceil}$.

As we noticed above, the translation does not use type annotations smartly. Indeed, all eliminations are explicitly typed and the translation could have been given in a weaker language with explicit elimination of polymorphism.

3.3 Encoding system F

Laüfer and Odersky have shown an encoding of system F into polymorphic datatypes [Odersky and Läufer, 1996]. 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 datatypes.

The types and the terms of system F are

$$t ::= \alpha \mid t \to t \mid \forall\alpha.t \qquad \text{Types}$$
$$M ::= x \mid M\,M \mid \lambda x{:}t.M \mid \Lambda\alpha.M \mid M\,t \qquad \text{Terms}$$

The translation of types of system F into types of our language is again straightforward, and may use a single label ε:

$$\langle\langle\alpha\rangle\rangle \stackrel{\triangle}{=} \alpha \qquad \langle\langle t_1 \to t_2\rangle\rangle \stackrel{\triangle}{=} \langle\langle t_1\rangle\rangle \to \langle\langle t_2\rangle\rangle \qquad \langle\langle\forall\alpha.t\rangle\rangle \stackrel{\triangle}{=} [\forall\alpha.\langle\langle t\rangle\rangle]^{\varepsilon}$$

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:

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

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

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

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: The proof is by structural induction on M. The only potential difficulty is to ensure that when typing $\langle a\rangle$ the polytype $[\sigma]^{\varepsilon}$ of a is always anonymous. This is immediate: since the translation of an abstraction is always annotated with the exact type of the variable, all type schemes of the typing environment may be fully generalized with respect to label variables; therefore there should be a derivation with no free labels in the typing environment where rule ELIM will always succeed. ■

If we choose for system F the semantics where abstraction does not stop evaluation, then the translation preserves the semantics in a strong sense (reduction steps of a term can be mapped to reduction steps of the translated term). Another semantics would need easy adjustment, either of the translation or of the semantics of our system.

The simplicity of our encoding of system F compared to its encoding into polymorphic datatypes [Odersky and Läufer, 1996] mainly results from having polytypes as first-class types. We have used a single label in the translation, as in the previous section. However, the derivation now relies on polymorphism of label variables in the construction $\lambda x{:}\tau.\,a$ and the elimination sites are left unannotated.

3.4 Comparison with system F

The above encoding shows that our system is a possible alternative to system F. Thus, it is interesting to compare a term M of system F with its translation a in our language, syntactically.

Our types differ by having an extra type constructor $[\_]$ surrounding every polymorphic type. Term variables do not carry type information in either M or a. Lambda abstractions carry exactly the same type information in both M and a. The type information at elimination of polymorphism is always omitted in a. The counterpart is that type information at introduction of polymorphism appears explicitly in $[a : \forall\alpha.\langle\langle\tau\rangle\rangle]$. In $\Lambda\alpha.M$, only the variable α is mentioned; the type τ is deduced from the type information located at abstraction and application nodes in M.

The comparison can be illustrated on the following two abstract examples:

$$\langle\langle\lambda(f{:}\forall\alpha.t).M\{f\,t'\}\rangle\rangle = \lambda f{:}[\forall\alpha.\tau].\,a\{\langle f\rangle\}$$

$$\langle\langle M_1\{\Lambda\alpha.M_2\}\rangle\rangle = a_1\{[a_2 : \forall\alpha.\tau]\}$$

The first example corresponds to the abstraction and use of a polymorphic value f in a function. Type annotations are similar in system F and our system, and we are even shorter since we can omit the instantiation types at polymorphism elimination. For such cases, our approach appears to be more comfortable than system F.

In the second example we introduce polymorphism somewhere inside a term. While system F can do it just by giving the type variable to quantify, we have to give an explicit polymorphic type. Indeed, our system provides no way to identify a type variable outside of an explicit type.

Which of the two syntaxes will be longer depends on which one of the two patterns dominates the other. We believe that the former pattern is more frequent in user programs, and that conversely the latter is more frequent in libraries. Hence, our system may provide a reasonable alternative syntax for higher-order programming.

Notice also that we have been comparing here a system F term and its direct translation in our system. Terms directly written in our system can omit much more type information. For instance, we do not actually need to provide a full type in our second example, but only a skeleton containing all occurrences of α in τ. And since we are extending ML, we do not need explicit type abstraction and instantiation for toplevel polymorphism.

We may also develop specific idioms. One of them is the simultaneous use of multiple type abstractions, 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.

This does not give us more concision than system F, but it allows to avoid the common pattern $[[f : \forall\alpha.\tau] : \forall\alpha'.\forall\alpha.\tau]$. In most cases, instantiation of all variables will be simultaneous and we can simply write $[f : \forall\alpha'.\forall\alpha.\tau]$.

3.5 Fully explicit type annotations

Considering the inherent difficulties of our semi-implicit elimination scheme, we present a sublanguage where elimination of polymorphism is fully explicit. This highlights the first stage of our proposal, *i.e.* making polymorphism explicit, while the second stage was dedicated to propagating type information.

This sublanguage is theoretically interesting. We do not lose any expressive power by enforcing explicit elimination of polymorphism, *i.e.* adding an explicit type annotation to all eliminations. Indeed, the encoding of polymorphic data types into polytypes has been done in such a restricted sublanguage. Simultaneously, the restriction to the sublanguage removes the need for labels, and hereby significantly simplifies the type system.

The encoding of system F is also possible and as easy in this restricted language. We just have to change the abstraction and type application rules.

$$\frac{A \vdash M : \forall\alpha.t' \Rightarrow a}{A \vdash M\,t : t'\{t/\alpha\} \Rightarrow \langle a : \langle\langle \forall\alpha.t' \rangle\rangle\rangle} \qquad \frac{A, x : t \vdash M : t' \Rightarrow a}{A \vdash \lambda x{:}t.M : t \to t' \Rightarrow \lambda x.a}$$

Changing the abstraction rule is not required, but annotating abstractions would be superfluous in this new translation. Notice however that terms encoded in the sublanguage are more verbose.

Finally, let us compare terms translated from our system into this restricted system. It looks like we would just have to move annotations from abstraction to elimination nodes, occasionally duplicating them. However, we see two main cases where this gets worse. Firstly, when an annotation contains several polytypes, like will often be the case for objects, we must split the annotation into pieces, and use a different type annotation to eliminate each polytype. Secondly, in our system we did not need any annotation at all for `let`-defined identifiers.

For a complete example, let $\tau_1 \stackrel{\triangle}{=} [\sigma_1]$, $\tau_2 \stackrel{\triangle}{=} [\sigma_2]$, $\tau_3 \stackrel{\triangle}{=} [\sigma_3]$ and $\tau \stackrel{\triangle}{=} \tau_1 \times \tau_2$. Using semi-implicit elimination we can write:

$$\texttt{let}\ y = [a : \sigma_3]\ \texttt{in}\ \lambda x{:}\tau.(\langle \texttt{fst}\ x\rangle, \langle \texttt{snd}\ x\rangle, \langle a\rangle)$$

but if we had only explicit elimination we would have to write:

$$\texttt{let}\ y = [a : \sigma_3]\ \texttt{in}\ \lambda x.(\langle \texttt{fst}\ x : \tau_1\rangle, \langle \texttt{snd}\ x : \tau_2\rangle, \langle a : \tau_3\rangle)$$

One could argue that some annotations in the second term are actually smaller than in the first one. We think however, that the number of annotations matters more than their size (which could always be shortened using type abbreviations).

In summary, restricting to fully-explicit polymorphism is interesting for its simplicity, but cannot stick syntactically to system F as much as semi-explicit polymorphism allows. It is also less convenient to use than the full system.

4 APPLICATION TO Objective ML

In this section we show how the core language can be used to provide polymorphic methods in Objective ML[5] [Rémy and Vouillon, 1997]. Polymorphic methods are useful in parameterized classes. Indirectly, they may also reduce the need for explicit coercions.

While Objective ML has parameterized 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` and it 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 sending a message to the object. The aim of the remainder of this section is to make invocation of polymorphic and monomorphic methods similar and also to make the invocation of polymorphic methods lighter.

The first step is to give polytypes to all methods. This is easily done by wrapping monomorphic methods into polytypes. For instance, we shall write

```
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 to the core language that solves this problem. We add a new typing rule ELIM-M:

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

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

Both rules ELIM and ELIM-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. The subject reduction property is also preserved.

[5] The examples of objects and classes given below are rather intuitive, and could be translated in other class-based object-oriented languages; the reader shall refer to [Rémy and Vouillon, 1997] for a

ELIM
$\textbf{let}\ \alpha \notin FV(A)\ \textbf{in}$
$\textbf{if}\ A \triangleright a : \alpha \Rrightarrow \exists\bar{\xi}.\,\theta\ \textbf{then}$
$\quad\textbf{if}\ \theta(\alpha) = [\forall\bar{\alpha}'.\tau']^{\varepsilon}\ \textbf{and}\ \varepsilon \notin FL(\theta(A))\ \textbf{then}\ A \triangleright \langle a\rangle : \tau \Rrightarrow \exists\bar{\xi},\alpha,\bar{\alpha}'.\,\theta\wedge\tau' = \tau$
$\quad\textbf{else if}\ \theta(\alpha) = \alpha'\ \textbf{and}\ \alpha' \notin FV(\theta(A))\ \textbf{then}\ A \triangleright \langle a\rangle : \tau \Rrightarrow \exists\bar{\xi},\alpha.\,\theta$
$\quad\textbf{else let}\ \varepsilon' \notin (FL(A)\cup FL(\tau))\ \textbf{in}\ A \triangleright \langle a\rangle : \tau \Rrightarrow \exists\bar{\xi},\varepsilon',\alpha.\,\theta\wedge\alpha \doteq [\tau]^{\varepsilon'}$
$\textbf{else}\ A \triangleright \langle a\rangle : \tau \Rrightarrow \bot$

Figure 5: Type inference rule for use of monomorphic polytypes

The expression $(\lambda x.\lambda y.\langle x\#\texttt{mem}\rangle\ y)$ is then typable with principal type $\langle\texttt{mem} : [\alpha \to \beta]; ..\rangle \to \alpha \to \beta$. Since all methods are now given polytypes, we shall 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.

formal presentation of Objective ML, allowing a deeper reading of this section.

Polymorphic methods also appear to be useful to limit the need for explicit coercions. In Objective ML, coercions are explicit. For instance, assume that objects of class point have the interface $\langle \mathtt{x} : \mathtt{int}; \mathtt{y} : \mathtt{int} \rangle$, 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 compute 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 $\langle \mathtt{x} : \mathtt{int}; \mathtt{y} : \mathtt{int}; \mathtt{color} : \mathtt{color} \rangle$) 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 $\langle \mathtt{x} : \mathtt{int}; \mathtt{y} : \mathtt{int}; \rho \rangle$. Here, `#point` contains a hidden row variable that is polymorphic in the function `distance`. This allows different applications to use different instances of the polymorphic 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, we must know here that `c` is a circle before using method `distance`, like would happen in more classical object-oriented type systems. There is an alternative between using explicit coercions or providing more type information. The advantage of type information is that it occurs at more convenient places. That is, it is necessary in method definitions and at the invocation of a method of 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.

5 VALUE-ONLY POLYMORPHISM

For impure functional programming languages, value-only polymorphism has become the standard way to handle the ubiquity of side-effects. It preserves type-soundness in the presence of side-effects, without making the type system overly complex. 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 [Wright, 1993].

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 the latter, since we also need rule GEN to precede rules ELIM and INTRO.

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

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

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

(LET-V)
$$\frac{A \vdash b : \varsigma \qquad A, x : \varsigma \vdash a : \tau}{A \vdash \mathtt{let}\; x = b \;\mathtt{in}\; a : \tau}$$

(LET-E)
$$\frac{A \vdash a_1 : \tau' \qquad A, x : \tau' \vdash a_2 : \tau}{A \vdash \mathtt{let}\; x = a_1 \;\mathtt{in}\; a_2 : \tau}$$

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):

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

This system works perfectly, and all properties are preserved.

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

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

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. There, polymorphism is easily recoverable 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:

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

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, its interest would be significantly reduced by this limitation.

One might think that allowing quantification on labels in LET-E, *i.e.* write $\forall\bar{\varepsilon}.\tau'$ in place of τ', is harmless. Indeed, label polymorphism does not allow type mismatches like usual polymorphism would: verifying identity of polymorphic types 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 the polytype $[\sigma]^{\varepsilon}\ \texttt{list}$ for any polymorphic type σ. However, the ordering of polymorphic types does not induce a corresponding ordering of polytypes, two polytypes with different polymorphic structure are unordered. Therefore, this expression has no principal type.

This problem is pathological, since such patterns will rarely occur. However, it is serious and significant machinery is required to fix it. It can be solved by restricting judgments to minimal ones. That is, we replace LET-V and LET-E by the restricted versions defined below. We write $A \vdash^{\star} a : \varsigma$ to mean that ς is a minimal type scheme for a under assumptions A. That is, $A \vdash a : \varsigma$ and there exists no ς' strictly greater than ς in the instantiation order, such that $A \vdash a : \varsigma'$. (Since we happen to be keeping principality, ς is the principal type scheme for a under assumptions A.)

(LET-V*)

$$\frac{A \vdash^{\star} b : \varsigma \qquad A, x : \varsigma \vdash a : \tau}{A \vdash \texttt{let}\ x = b\ \texttt{in}\ a : \tau}$$

(LET-E*)

$$\frac{A \vdash^{\star} a_1 : \forall\bar{\varepsilon}\bar{\alpha}.\tau' \qquad a_1 \not\equiv b \qquad A, x : (\forall\bar{\varepsilon}.\tau')\{\bar{\tau}/\bar{\alpha}\} \vdash a_2 : \tau}{A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau}$$

The rule LET-E* may seem strange, since it is not an instance of the original LET rule, but rather a combination of INST and LET. The original derivation would have been:

$$\frac{\dfrac{\dfrac{A \vdash a_1 : \forall\bar{\varepsilon}\bar{\alpha}.\tau'}{\vdots}}{A \vdash a_1 : \forall\bar{\varepsilon}.\tau''} \qquad A, x : \forall\bar{\varepsilon}.\tau'' \vdash a_2 : \tau}{A \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau}$$

The restriction to minimal judgments is not new: it has already been used for the typing of dynamics in ML [Leroy and Mauny, 1991], for instance. One has to reject the program $\lambda x.(\texttt{dynamic}\ x)$ because, in the principal judgment $x : \alpha \vdash x : \alpha$, some variable of the type of x occurs free in the context. A non principal judgment obtained by choosing `int` for α would be correct, but arbitrary. More recently, it has been used for local type inference in system $F_{\leq}$ [Pierce and Turner, 1998].

Type inference is only allowed locally at application nodes, and upon the condition there is a principal solution to the local inference problem. Without this condition, choices made at an application node would influence other nodes, and inference would lose its locality.

We use minimality here in a somewhat different way. In the above two systems, requiring a principal solution was a way to have the inference fail on some ambiguous cases. Contrary to dynamics, our types do not need to be ground; they may share variables with the environment. Contrary to local type inference, all our satisfiable inference problems have principal solutions. Thus, our minimality condition never makes a type inference problem fail, but only restricts the set of types that can be assigned to a variable in a let statement. Notice that $\vdash^\star$ judgments do not actually require the derivation to be principal, but only minimal; they do not eliminate all different derivations, but only those that would be obtained by unnecessarily instantiating some types. We may then prove the existence of principal types by showing that all minimal type schemes are equal modulo renaming of bound variables, and as a result our minimality condition happens to be a principality condition. This condition is not harmful when reasoning about derivations: the property of minimality of a derivation is kept by global substitution of free type variables, so that the stability lemma is still valid in the extended system.

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.

6 RELATED WORK

Full type inference of polymorphic types is undecidable [Wells, 1994]. 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 [Cardelli, 1993] polymorphic types are marked either as implicit (actually their variables are marked) and they are automatically instantiated when used, or as explicit and they remain polymorphic until they are explicitly instantiated. 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 annotation on the variable x, which shows that this is not an extension to ML. Pierce and Turner have extended this partial inference mechanism to $F^{\omega}_{\leq}$ in the design of the language Pict [Pierce and Turner, 1997b]. 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.

Conscious of this problem, they more recently proposed to replace this unpredictable approach by one based on predictable local inference [Pierce and Turner, 1998, Pierce and Turner, 1997a]. Their approach is somewhat opposite of ours: while we provide some inference-free type checking without modifying ML type inference, they add some type inference to $F_{\leq}$ and keep a checking based system. In their approach, the uniqueness of typing is still valid at every step. As we, they distinguish between the specification and the algorithm of type inference, but this distinction is only limited to one rule, the one doing local inference. This rule has two provably equivalent versions: one is a specification of the inferred type in terms of a universal property; the other one is algorithmic and is presented in a constraint-solving style. The difference of approach and the fact that they also handle subtyping make it difficult to compare the respective strength of the two systems.

A different approach is taken by Pfenning [Pfenning, 1988]. 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 [Pfenning, 1993]. As in our system, his solution is an extension of ML. His solution is also more powerful than ours; the price is the loss of principal types and decidability of type inference. A decidable subcase of higher-order unification has also been considered in [Dowek et al., 1996]. Neither solution handles subtyping yet.

Kfoury and Wells show that type inference could be done for the rank-2 fragment of system F [Kfoury and Wells, 1994]. However, they do not have a notion of principal types. It is also unclear how partial type information could be added.

In [Odersky and Läufer, 1996], 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 in section 3.2, our framework subsumes theirs. They also introduce another mechanism that allows annotations of abstractions by polymorphic types as in $\lambda x\!:\!\sigma.x$ together with a type containment relation on polymorphic types similar to the one of Mitchell [Mitchell, 1984] but with some serious restriction. Polymorphic types may be of the form $\forall\alpha.\sigma_1 \to \sigma_2$, where σ_i are polymorphic types themselves. However, universal variables such as α can only be substituted by simple types. Thus, the only way to apply a function of type $\forall\alpha.\alpha \to \alpha$ to a polymorphic value remains to embed the argument inside an explicitly defined polytype. Actually, one of the reasons for complementing universal-datatype polymorphism by restricted type-containment is to obtain an encoding of system F. In our case, the encoding of system F is permitted by the use of polytypes.

In [Duggan, 1995], 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 having fully explicit polymorphism both at introduction and elimination, as in [Odersky and Läufer, 1996]. 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

We have presented a conservative extension to ML that allows for first-class polytypes and first-class polymorphic values. In our proposal as in ML, let-polymorphism remains implicit. While first-class polymorphism must be introduced explicitly, type information is inferred at the elimination point. This allows for polymorphic methods in Objective ML, which are particularly useful in parameterized classes.

We have also shown that polymorphism can be restricted to values, so as to be sound in the presence of side-effects. This naive standard restriction weakens the propagation of first-class polymorphism, and forces some type annotations, unnecessarily. Thus, we have also proposed an extension that covers all useful cases and does not present any known limitations. Even though the specification of typechecking becomes technically more difficult, since it involves the notion of minimal judgments, the principal-type property is preserved. Although practically insignificant, this difficulty exposes a drawback of the value-only restriction of polymorphism.

As future work, three extensions of importance are to be studied. While second order polymorphism is sometimes quite useful for programming, it is not always enough. Indeed, this is only one step further on the scale of abstraction. There are few but serious situations when system F^{ω} is needed to accomplish the desired abstraction. Extending our solution to F^{ω} might be possible, but certainly trickier because of β-reduction at the level of types. Secondly, we should consider applying our technique to existential types. The encoding of these into universal types introduces inner quantifiers, which removes all opportunities for inference. It remains unclear whether primitive existential types could benefit from our work. Thirdly, the replacement of the core ML type system by one with subtyping constraints as in [Aiken and Wimmers, 1993, Eifrig et al., 1995a], would combine first-order generic polymorphism and subtyping polymorphism in an ML-like language. The issues of constraint checking and type generalization are rather orthogonal. However, some recent and more general presentation [Pottier, 1996, Eifrig et al., 1995b] significantly differs from ML. Thus, more investigation is required.

The principle of our approach has been to keep within first-order type inference. While we believe this to be sufficient in practice, we would still like to formulate our type system in terms of partial type inference for second-order lambda-calculus.

References

[Aiken and Wimmers, 1993] Aiken, A. and Wimmers, E. L. (1993). Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture*, pages 31–41. ACM press.

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

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

[Dowek et al., 1996] Dowek, G., Hardin, T., Kirchner, C., and Pfenning, F. (1996). Higher-order unification via explicit substitutions: the case of higher-order patterns. In Maher, M., editor, *Joint international conference and symposium on logic programming*, pages 259–273.

[Duggan, 1995] Duggan, D. (1995). Polymorphic methods with self types for ML-like languages. Technical report cs-95-03, University of Waterloo.

[Eifrig et al., 1995a] Eifrig, J., Smith, S., and Trifonov, V. (1995a). Sound polymorphic type inference for objects. In *OOPSLA*.

[Eifrig et al., 1995b] Eifrig, J., Smith, S., and Trifonov, V. (1995b). Type inference for recursively constrained types and its application to OOP. In *Mathematical Foundations of Programming Semantics*.

[Garrigue and Rémy, 1997] Garrigue, J. and Rémy, D. (1997). Extending ML with semi-explicit higher-order polymorphism. In Ito, T. and Abadi, M., editors, *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 20–46. Springer-Verlag.

[Huet, 1976] Huet, G. (1976). *Résolution d'équations dans les langages d'ordre* $1,2,\ldots,\omega$. Thèse de doctorat d'état, Université Paris 7.

[Jouannaud and Kirchner, 1991] Jouannaud, J.-P. and Kirchner, C. (1991). Solving equations in abstract algebras: a rule-based survey of unification. In Lassez, J.-L. and Plotkin, G., editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. MIT-Press, Cambridge (MA, USA).

[Kfoury and Wells, 1994] Kfoury, A. J. and Wells, J. B. (1994). 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.

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

[Leroy and Mauny, 1991] Leroy, X. and Mauny, M. (1991). Dynamics in ML. In Hughes, J., editor, *Conference on Functional Programming and Computer Achitecture*, volume 523 of *Lecture Notes in Computer Science*, pages 406–426. Springer-Verlag.

[Mitchell, 1984] Mitchell, J. C. (1984). 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. 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.

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

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

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

[Pierce and Turner, 1997a] Pierce, B. C. and Turner, D. N. (1997a). Local type argument synthesis with bounded quantification. Technical Report 495, Computer Science Department, Indiana University.

[Pierce and Turner, 1997b] Pierce, B. C. and Turner, D. N. (1997b). Pict: A programming language based on the pi-calculus. Technical report, Computer Science Department, Indiana University.

[Pierce and Turner, 1998] Pierce, B. C. and Turner, D. N. (1998). Local type inference. In *Proceedings of the 25th ACM Conference on Principles of Programming Languages*. Full version available as Indiana University CSCI Technical Report 493.

[Pottier, 1996] Pottier, F. (1996). Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 122–133.

[Rémy, 1992] Rémy, D. (1992). 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.

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

[Rémy and Vouillon, 1997] Rémy, D. and Vouillon, J. (1997). 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.

[Wells, 1994] Wells, J. B. (1994). 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.

[Wright, 1993] Wright, A. K. (1993). Polymorphism for imperative languages without imperative types. Technical Report 93–200, Rice University.

A PROOFS OF MAIN THEOREMS

Lemmas 1 (*canonical derivations*) and 2 (*stability by substitution*) are tedious but essential in ML. Their proofs easily carry over with the three new rules, ANN, INTRO, and ELIM.

Proof of type soundness for the core language

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

Proof: The proof is by induction on the structure of the first derivation. ■

Theorem 1 (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 but CONTEXT, we assume that $A \vdash a_1 : \varsigma$ (1) and that $a_1 \longrightarrow a_2$, (the structure of a_1 depending on the case) and we show that $A \vdash a_2 : \varsigma$ (2).

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

$$\cfrac{\cfrac{\Delta}{A \vdash a_1 : \varsigma_0}}{A \vdash a_1 : \forall \bar{\xi}.\varsigma_0} \; (\textsc{Gen}*)$$

where the derivation Δ of (1) does not end with a rule GEN. Thus we have $A \vdash a_2 : \varsigma_0$ and (2) follows by the same sequence of generalizations.

Case FUN **and** LET**:** This is a straightforward application of term-substitution lemma.

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

$$\cfrac{\cfrac{\cfrac{A \vdash a : \sigma_1 \qquad (\sigma_1 : \sigma_0 : \sigma_2)}{A \vdash [a : \sigma_0] : [\sigma_2]^\varepsilon} \; (\textsc{Intro})}{A \vdash [a : \sigma_0] : \forall \varepsilon.[\sigma_2]^\varepsilon} \; (\textsc{Gen})}{A \vdash \langle [a : \sigma_0] \rangle : \sigma_2} \; (\textsc{Elim})$$

The polymorphic types 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 variables $\bar{\alpha}$ that do not occur free in A, we can contract this derivation into the following derivation of (2):

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

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

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

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

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

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

$$\dfrac{\dfrac{A \vdash a : \sigma_1'\ (3) \qquad (\sigma_1' : \sigma_1 : \sigma_1'')\ (4)}{A \vdash [a : \sigma_1] : [\sigma_1'']^{\varepsilon_1}}\,(\textsc{Intro}) \qquad ([\sigma_1'']^{\varepsilon_1} : [\sigma_2]^{\varepsilon_2} : [\sigma_3]^{\varepsilon_3})\ (5)}{A \vdash ([a : \sigma_1] : [\sigma_2]^{\varepsilon_2}) : [\sigma_3]^{\varepsilon_3}}\,(\textsc{Ann})$$

Let $\forall\bar{\alpha}.\tau_1$ be σ_1. From (4), we know that we can write σ_1' and σ_1'' as $\forall\bar{\alpha}.\tau_1'$ and $\forall\bar{\alpha}.\tau_1''$. Moreover, we have $(\tau_1' : \tau_1 : \tau_1'')$. From (5), we also get $(\sigma_1'' : \sigma_2 : \sigma_3)$. Thus, we have

$$\dfrac{\dfrac{\dfrac{\dfrac{(3)}{A \vdash a : \tau_1'}\,(\textsc{Inst*}) \qquad (\tau_1' : \tau_1 : \tau_1'')}{A \vdash (a : \tau_1) : \tau_1''}\,(\textsc{Ann})}{A \vdash (a : \tau_1) : \sigma_1''}\,(\textsc{Gen*}) \qquad (\sigma_1'' : \sigma_2 : \sigma_3)}{A \vdash [(a : \tau_1) : \sigma_2] : [\sigma_3]^{\varepsilon_3}}\,(\textsc{Intro})$$

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

Case CONTEXT**:** Here, we need to show that if $a_1 \subset a_2$ then for any evaluation context E we also have $E\{a_1\} \subset E\{a_2\}$. The proof, which we can directly take from usual ML, is by structural induction on E.

■

Theorem 2 (Canonical forms) *Irreducible programs (for call-by-value reduction) 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 all possible canonical derivations, we see that:

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

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

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

Then, the theorem follows: considering a program a that is well-typed in the empty environment and that cannot be reduced, it can easily be shown by structural induction that a is a value. ■

Proof of the principal type property

Lemma 5 (Unification) *Each of the rules given in figures 2 and 3, is correct and complete.*

Proof:

Cases OCCUR-CHECK**,** MERGE**,** ABSORB**, and** DECOMPOSE**:** those are standard rules for first-order unification.

Case DECOMPOSE-POLY **and** CLASH**:** immediate.

Case POLYTYPES**:** This case amounts to fully formalizing the discussion in section 2.4. Assume that $\bar{\alpha} \cap \bar{\alpha}'$, $\bar{\alpha} \cap FV(\tau')$, and $\bar{\alpha}' \cap FV(\tau)$ are all empty (1).

Soundness: Assume that θ is a solution of $\exists\,\bar{\alpha}\bar{\alpha}'.\tau \doteq \tau' \wedge \bar{\alpha} \leftrightarrow \bar{\alpha}'$. Let η be a renaming of $\bar{\alpha}\bar{\alpha}'$ into variables outside of free variables of θ, τ, τ', and $\bar{\alpha}\bar{\alpha}'$. The substitution $\eta \circ \theta$ is also a solution of the same unificand. Since its image has no variable in common with $\bar{\alpha}\bar{\alpha}'$, the substitution $\eta \circ \theta \setminus \bar{\alpha}\bar{\alpha}'$ can be extended by a substitution ρ of domain $\bar{\alpha}\bar{\alpha}'$ such that the substitution θ' equal to $\eta \circ \theta \setminus \bar{\alpha}\bar{\alpha}' + \rho$ is a solution of $\tau \doteq \tau' \wedge \bar{\alpha} \leftrightarrow \bar{\alpha}'$. Since θ' is a solution of $\bar{\alpha} \leftrightarrow \bar{\tau}'$, the substitution ρ is injective on $\bar{\alpha}$ and $\bar{\alpha}'$ taken separately. Moreover, its image is in $\bar{\alpha}\bar{\alpha}'$. The substitution $(\eta + \eta^{-1}) \circ \theta'$ decomposes as $(\theta \setminus \bar{\alpha}\bar{\alpha}') + (\eta \circ \rho)$, which is actually equal to $\theta \circ \eta \circ \rho$; it must be a solution of $\tau \doteq \tau'$. Therefore the substitution θ is a solution of $\forall\bar{\alpha}.\tau \stackrel{\forall}{=} \forall\bar{\alpha}'.\tau'$.

Completeness: Let θ be a solution of $\forall\bar{\alpha}.\tau \stackrel{\forall}{=} \forall\bar{\alpha}'.\tau'$. Reusing the reasoning and the definitions of section 2.4, the substitution $(\eta \circ \theta \setminus \bar{\alpha}\bar{\alpha}') + \rho$ is a solution of $\tau \doteq \tau' \wedge$

$\bar\alpha \leftrightarrow \bar\alpha'$ where η is a renaming of $\bar\alpha\bar\alpha'$ into variables taken outside of free variables of θ, τ, τ', and $\bar\alpha\bar\alpha'$. Thus, $\eta \circ \theta$ is a solution of $\exists\, \bar\alpha\bar\alpha'.\, \tau \doteq \tau' \wedge \bar\alpha \leftrightarrow \bar\alpha'$ and so is θ by composition with η^{-1}.

Case RENAMING-TRUE**:** The completeness is obvious. For the soundness, let θ be any substitution. Let η be a renaming of $\bar\alpha\bar\alpha'$ outside of $\bar\alpha\bar\alpha'$ and free variables of θ. The substitution $(\eta \circ \theta) \setminus \bar\alpha\bar\alpha'$ can be extended with the substitution $(\alpha_i \mapsto \alpha'_i)^{i\in 1..n}$. Clearly, this extension satisfies both $(\alpha_i \doteq \alpha'_i)^{i\in 1..n}$ and $\bar\alpha \leftrightarrow \bar\alpha'$. Thus θ is a solution of $\exists\, \bar\alpha\bar\alpha'.\, (\alpha_i \doteq \alpha'_i)^{i\in 1..n} \wedge \bar\alpha \leftrightarrow \bar\alpha'$.

Case RENAMING-FALSE**:** The soundness is obvious. For the completeness let us consider the two following cases:

$\beta \in \bar\alpha$ **and** $\tau \notin \bar\alpha' \cup \{\beta\}$: Assume that there exists a solution θ of both $\beta \doteq \tau \doteq e$ and $\bar\alpha \leftrightarrow \bar\alpha'$. Since $\theta(\tau)$ is equal to $\theta(\beta)$, it must be a variable, and so should τ itself. Since $\theta \setminus \bar\alpha\bar\alpha'$ should not have variables in common with $\bar\alpha\bar\alpha'$, τ must be in $\bar\alpha\bar\alpha'$. However, since it is not in $\bar\alpha'$, it must be another variable γ of $\bar\alpha$ distinct from β, which contradicts with the fact that $\theta \upharpoonright \bar\alpha$ must be injective (condition 2).

$\beta \in \bar\alpha \cap FV(\tau)$ **and** $\tau \neq \beta$: In particular, τ must be a proper term. Assume that there exists a solution θ of both $\gamma \doteq \tau \doteq e$ and $\bar\alpha \leftrightarrow \bar\alpha'$. The term $\theta(\gamma)$, equal to $\theta(\tau)$, is a proper term; thus, γ cannot be a variable of $\bar\alpha\bar\alpha'$. However, $\theta(\gamma)$ contains the variable $\theta(\beta)$ that belongs to $\bar\alpha\bar\alpha'$. This contradicts condition 3.

■

Theorem 4 *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, or there is no solution and the problem reduces to $\bot$.*

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 INTRO**:** We assume that all the conditions of the first four lines are satisfied. We write σ_i for $\sigma\{\bar\alpha_1\bar\varepsilon_i/\bar\alpha_0\bar\varepsilon_0\}$.

Soundness: Let us assume that $A \vdash a : \tau_1\{\bar\varepsilon_1/\bar\varepsilon_0\} \Rrightarrow \exists\xi.\theta$ and $\bar\alpha \cap dom(\theta) \cup FV(codom(\theta)) = \emptyset$. We have $\theta(A) \vdash a : \theta(\sigma_1)$ by generalization of $\bar\alpha$ in the judgment $\theta(A) \vdash a : \theta(\tau_1\{\bar\varepsilon_1/\bar\varepsilon_0\})$. Since by construction $(\theta(\sigma_1) : \sigma : \theta(\sigma_2))$, we also have $\theta(A) \vdash [a : \sigma] : \theta([\sigma_2]^\varepsilon)$. That is, θ is a solution of $A \vdash [a : \sigma] : [\sigma_2]^\varepsilon$. Thus, a solution of $\theta \wedge \tau = [\sigma_2]^\varepsilon$ is a solution of $A \vdash [a : \sigma] : \tau$. Moreover, no variable of $\bar\varepsilon_1, \bar\varepsilon_2, \varepsilon, \bar\alpha_1$ appears in A or τ.

Completeness: Let us assume that θ' is a solution of $A \triangleright [a : \sigma] : \tau$. A canonical derivation of $\theta'(A) \vdash [a : \sigma] : \theta'(\tau)$ must end with rule INTRO. Thus, there exists some polymorphic types σ'_1 and σ'_2 and some label ε such that $\theta'(A) \vdash a : \sigma'_1$ (1), $(\sigma'_1 : \sigma : \sigma'_2)$ (2), and $\theta'(\tau) = [\sigma'_2]^\varepsilon$ (3). By definition of the relation $(\_ : \sigma : \_)$ the pair (σ'_1, σ'_2) must be of the form $(\theta''(\sigma_1), \theta''(\sigma_2))$ for some substitution θ'' of domain $\bar\varepsilon_1 \bar\varepsilon_2 \bar\alpha_0$. A canonical derivation of (1) must end with a succession of rules GEN. Thus we have $\theta'(A) \vdash a : \theta''(\tau_1\{\bar\varepsilon_1/\bar\varepsilon_0\})$. On the one hand, the substitution $\theta' + \theta''$ is a solution of $A \vdash a : \tau_1\{\bar\varepsilon_1/\bar\varepsilon_0\}$, and consequently a solution of θ. On the other hand, it is a solution of $\tau = [\tau_1\{\bar\varepsilon_2/\bar\varepsilon_0\}]^\varepsilon$. Moreover, it extends θ' on $\bar\alpha_0$, $\bar\varepsilon_0$, $\bar\varepsilon_1$, ε, and $\bar\xi$.

The completeness of the else branch is straightforward; The proof above actually applies if θ is $\perp$. If θ is not $\perp$, the right condition may always be satisfied since $\bar\alpha$ is disjoint from free variables of the typing problem.

Case ELIM**:** We assume that the condition of the first line is satisfied.

Soundness: If $\theta(\alpha) = [\forall\bar\alpha'.\tau']^\varepsilon$ and $\varepsilon \notin FL(\theta(A))$ then rule ELIM applies, and an extension of θ such that $\theta(\tau) = \theta(\tau')$ is 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 $\theta(A) \vdash a : [\tau]^\varepsilon$ for some ε not in $FV(\theta(A))$. By generalization of ε and rule ELIM, 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 ELIM. Thus, we must have $\theta'(A) \triangleright a : [\sigma]^{\varepsilon'}$ for some ε' that does not appear in $\theta'(A)$ and some polymorphic type 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 $\theta \wedge \theta(\alpha) \doteq [\sigma]^{\varepsilon'}$ (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 $\theta(\alpha) = [\forall\bar\alpha'.\tau']^\varepsilon$ then ε cannot belong to $FL(\theta(A))$, otherwise ε' would belong to $FL(\theta'(A))$. Since θ' is a solution of $[\sigma]^{\varepsilon'} \doteq [\forall\bar\alpha'.\tau']^\varepsilon$, 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 of the lexicographic ordering composed of successively

1. the sum of sizes of program components,
2. the sum of monomials $X^{size(\sigma)}$ for all type and polymorphic type components of the system,
3. the number of polymorphic constraints,

4. the number of multi-equations,
5. the sum of the lengths of multi-equations, and
6. the number of renaming problems.

Moreover, unification problems that cannot be reduced are in solved form. Clearly, there cannot remain any typing problem since for each construction of the language some rule applies. Similarly, polytypes can always be decomposed. Let us consider a renaming problem $\bar{\alpha} \leftrightarrow \bar{\alpha}'$ for which rule RENAMING-FALSE would not apply. Then variables of $\bar{\alpha}\bar{\alpha}'$ could only appear in multi-equations composed of the variables in $\bar{\alpha}\bar{\alpha}'$. Moreover at most one variable of each set $\bar{\alpha}$ and $\bar{\alpha}'$ could appear in each of these multi-equations. Therefore rule RENAMING-TRUE would apply. The remaining rules are standard rules for unification for simple types. ∎