Simple, partial type-inference for System F based on type-containment

Didier Rémy

To cite this version:

Didier Rémy. Simple, partial type-inference for System F based on type-containment. Proceedings of the tenth International Conference on Functional Programming, Sep 2005, Tallinn, Estonia. inria-00000938

HAL Id: inria-00000938

https://hal.inria.fr/inria-00000938

Submitted on 15 Dec 2005

Simple, partial type-inference for System F based on type-containment

Didier Rémy

INRIA-Rocquencourt

http://pauillac.inria.fr/~remy

Abstract

We explore partial type-inference for System F based on type-containment. We consider both cases of a purely functional semantics and a call-by-value stateful semantics. To enable type-inference, we require higher-rank polymorphism to be user-specified via type annotations on source terms. We allow implicit predicative type-containment and explicit impredicative type-instantiation. We obtain a core language that is both as expressive as System F and conservative over ML. Its type system has a simple logical specification and a partial type-reconstruction algorithm that are both very close to the ones for ML. We then propose a surface language where some annotations may be omitted and rebuilt by some algorithmically defined but logically incomplete elaboration mechanism.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Polymorphism

General Terms Design, Reliability, Languages, Theory, Verification

Keywords Type Inference, System F, Polymorphism, Type Reconstruction, Type Containment, Elaboration

Introduction

ML-style polymorphism has often been considered as a local optimal, offering one of the best compromises between simplicity and expressiveness. It is at the heart of two successful families of languages, Haskell and ML. In the last two decades, both Haskell and ML type systems evolved in surprisingly different directions while retaining their essence.

However, programmers are starting to feel restricted as programs become bigger and advanced programming patterns are used. The demand for first-class polymorphism has been increasing. First-class polymorphism is not only sometimes useful—it is quickly becoming unavoidable!

The reference calculus for first-class polymorphism is System F. Unfortunately, full type inference is undecidable in System F [Wel94]. Adding first-class polymorphism to ML should not sacrifice type inference, one of the attractive aspects of ML. One approach to keeping type inference is to reduce it to second-order unification [Pfe88]. However, even so only provides with a semi-algorithm. Moreover, unintuitive explicit marks for type abstractions and applications are still required.

An alternate approach is to explicitly annotate source terms with their types, as in Church's presentation of System F. This turns type inference into a simple type checking algorithm. However, type annotations are quickly a burden to write and often become obtrusive, even for simple ML programs.

Of course, one wishes to have simultaneously the expressiveness of System F, the decidability of its explicitly typed version, and the *convenience* of ML-like type inference, if not full type inference. The idea is thus to bring System F and ML closer. This can be approached from two opposite directions. Starting with explicitly typed System F, one may perform some partial type inference so as to alleviate the need for some (but not all) type annotations. Or conversely, starting with ML, one may allow some explicit type annotations so as to reach most or all of System F programs.

The first approach is known as local type inference [PT00, OZZ01]. By definition, these solutions cover all of System F. However, they are not yet conservative over ML, that is, there remain ML programs that fail to type without annotations. In fact, local type inference allows getting rid of many silly and annoying type annotations, but fails to make *all* of them redundant, so some unintuitive annotations remain needed [HP99]. Here, we focus on the second approach. By construction, it leads to conservative extensions of ML.

The simplest method to extend ML with first-class polymorphism is to use *boxed* polymorphism. The idea is to embed first-class polymorphic types into simple types using explicit injection and projection functions. These coercions can be automatically attached to data constructors via algebraic data type definitions, which makes them simple and sometimes transparent to the user. This solution was originally proposed for existential types [LO94] and then applied to universal types [Rém94]. Boxed polymorphism is extremely simple, because ML never sees second-order types. The drawback is that boxes are rigid: they need to be declared and explicitly inserted. Furthermore, the history of the construction of a polymorphic value is recorded in its type, as the stacking of boxes.

Odersky and Laüfer [OL96] later observed that some types need not be boxed. They proposed a type system where the user can write $\lambda \mathsf{z} : \forall \alpha. \alpha \to \alpha . \mathsf{t}$, making the λ-bound variable z available within t at type $\forall \alpha. \alpha \to \alpha$, as if it were let-bound. Another key feature in the work of Odersky and Laüfer is to generalize the instance relation of ML, allowing instantiation of toplevel quantifiers as in ML but also of inner quantifiers by recursively traversing arrow types, co-variantly on the right-hand side to instantiate their codomains and contra-variantly on the left-hand side to generalize their domain. This allows to keep types as polymorphic as possible and only instantiate them by need. For instance, $\lambda \mathsf{z} : \forall \alpha. \alpha \to \alpha . \lambda \mathsf{y} . \mathsf{z}$ may be typed as $\forall \gamma.(\forall \alpha. \alpha \to \alpha) \to \gamma \to (\forall \alpha. \alpha \to \alpha)$ but still used with type $\forall \gamma.(\forall \alpha. \alpha \to \alpha) \to \gamma \to (\forall \beta. (\beta \to \beta) \to (\beta \to \beta))$. There is a restriction, however, that is essential to allow simple type-inference: when a type variable such as α above is instantiated, it can only be replaced with a monotype τ. This built-in restriction to predicative polymorphism is actually quite severe, as we shall see. Fortunately, it can be circumvented by keeping boxed types, which allow more explicit but impredicative type instantiation, coexisting with implicit predicative polymorphism [OL96].

Peyton-Jones *et al.* have recently extended Odersky and Laüfer's proposal to propagate type annotations in source programs [PVWS05a], so that an external type annotation, such as $\lambda \mathsf{z} . \mathsf{t} : (\forall \alpha. \alpha \to \alpha) \to \tau$, behaves as if the argument z was also annotated with type $\forall \alpha. \alpha \to \alpha$. Their type system, which we hereafter refer to as PVWS, has been ingeniously tuned. It is also quite involved. The authors claim that: *adding higher-rank polymorphism to ML is so simple that every implementation of ML-style type inference should have it.* They argue that very few changes need to be made to adapt an implementation of ML type inference to predicative higher-rank polymorphism. While we accept their claim, we find their implementation-based demonstration unconvincing and their specification too algorithmic and unintuitive. Indeed, PVWS mixes several features that are often considered conflicting: (1) ML-like type inference; (2) a form of contravariant subsumption, and (3) propagation of user-provided annotations. Type systems that mix (1) and (2) usually exploit explicit constraints to keep principal types, which PVWS does not; (3) enforces a strict order in which type checking must be performed, which usually stand in the way when inferring principal types. Such an unusual combination of features does not imply misconception, of course—and indeed, PVWS has been carefully designed. However, it should then be studied with a lot of attention [PVWS05b].

Contributions

In this paper, we investigate partial type inference based on type containment and first-order typing constraints. Our primary goal is to bring further evidence to Peyton-Jones and Shields' claim and thus to provide more insight to PVWS. Another goal is also to explore the design space and, in particular, how far one can go within an ML-like type-inference framework. We have at least four different contributions: A preliminary, independent result is the soundness of a variant of F^η for a call-by-value stateful semantics. Our main contribution is the core language F_{ML} that brings down System F to the level of ML—it has a simple logical specification and a sound and complete first-order type inference algorithm. A secondary contribution is the surface language $\mathsf{F}^?_{\mathsf{ML}}$ and our proposal to split partial type inference in $\mathsf{F}^?_{\mathsf{ML}}$ into a composition of two separate orthogonal phases: an algorithmic elaboration process into F_{ML}, followed by type inference into F_{ML}. Our approach leaves room for variations in the definition of type-containment relations, which controls the expressiveness of the core language, and in the elaboration process. A side contribution is also the exploration of the design space: our two-phase decomposition has advantages but also some limitations.

1. System $\mathsf{F}(\leqslant)$ and some instances

System F is the canonical system for λ-calculus with second-order polymorphism. However, it may be presented in two ways. In Church's view, terms are explicitly typed, including type annotations on λ-abstractions, and type applications. Therefore, type-checking only needs to verify that user-provided type information is correct with respect to the typing rules. On the opposite, in Curry's view, terms are untyped and type-checking must infer the types that should be assigned to formal parameters in λ-abstractions as well as the places where type abstractions and type applications should be inserted. The two views can be reconciled by considering type inference for terms in Curry's style as *elaborating* an explicitly typed term in Church's style.

1.1 Terms and types

We assume given a finite or denumerable collection of constants (ranged over by c) and a denumerable collection of variables (ranged over by z and y). Constants and variables form identifiers, which we write x. Terms, ranged over by t, are those of the untyped λ-calculus, *i.e.*, identifiers x, abstractions $\lambda \mathsf{z} . \mathsf{t}$, or applications $\mathsf{t}\ \mathsf{t}'$. Application has higher priority than abstraction and is left-associative *i.e.* $\lambda \mathsf{z} . \lambda \mathsf{z}' . \mathsf{t}\ \mathsf{t}'\ \mathsf{t}''$ stands for $\lambda \mathsf{z} . (\lambda \mathsf{z}' . ((\mathsf{t}\ \mathsf{t}')\ \mathsf{t}''))$.

We assume a denumerable collection of type variables (ranged over by α). Types, ranged over by σ, are type variables, arrow types $\sigma_1 \to \sigma_2$, and polymorphic types $\forall \alpha. \sigma$. The $\forall$ symbol acts as a binder for α in σ. Free type variables in a type σ, which we write $\mathsf{ftv}(\sigma)$, are defined accordingly. We always consider types equal modulo renaming of bound-type variables. The scope of $\forall$-quantification extends to the right as much as possible and arrows are right-associative. That is, $\forall \alpha. \sigma_1 \to \sigma_2 \to \sigma_3$ stands for $\forall \alpha. (\sigma_1 \to (\sigma_2 \to \sigma_3))$.

A type variable α occurs positively in α. If α occurs positively (resp. negatively) in σ, then it occurs positively (resp. negatively) in $\sigma' \to \sigma$ and $\forall \beta. \sigma$ when β is distinct from α, and negatively (resp. positively) in $\sigma \to \sigma'$. We write $\mathsf{ftv}^+(\sigma)$ (resp. $\mathsf{ftv}^-(\sigma)$) the sets of type variables that occur positively (resp. negatively) in σ. We write $\mathsf{ftv}^\dagger(\sigma)$ the set $\mathsf{ftv}^+(\sigma) \setminus \mathsf{ftv}^-(\sigma)$, which we call *non-negative free type variables.*

A relation $\mathcal{R}$ on types is *structural* if it is reflexive, transitive and closed under the following properties: if $\sigma_1\ \mathcal{R}\ \sigma_2$, then $\forall \alpha. \sigma_1\ \mathcal{R}\ \forall \alpha. \sigma_2$ (S-All), $\sigma_2 \to \sigma\ \mathcal{R}\ \sigma_1 \to \sigma$ (S-Contra) and $\sigma \to \sigma_1\ \mathcal{R}\ \sigma \to \sigma_2$ (S-Cov). A *congruence* is a structural equivalence relation.

Let $\approx$ be the smallest congruence that allows commutation of adjacent binders, *i.e.* $\forall \alpha. \forall \beta. \sigma \approx \forall \beta. \forall \alpha. \sigma$, and removal of redundant binders, *i.e.* $\forall \alpha. \sigma \approx \sigma$ whenever α is not free in σ. Most relations on types we will consider will be subrelations of $\approx$. Therefore, we could usually treat types equal modulo $\approx$. However, we prefer not to do so and treat $\approx$ explicitly when needed. A canonical form for $\approx$-equivalence is one without redundant quantifiers and where adjacent quantifiers are listed in order of apparition. Canonical forms are unique (up to renaming of bound-type variables, but we treat such types as equal). We write $\mathsf{canon}(\sigma)$ for the canonical form of σ. Subterms of types in canonical forms are also in canonical form.

Figure 1. Typing rules for $\mathsf{F}(\leqslant)$

$$\textsc{Var}\quad \frac{\mathsf{x} : \sigma \in \Gamma}{\Gamma \vdash \mathsf{x} : \sigma} \qquad \textsc{Inst}\quad \frac{\Gamma \vdash \mathsf{t} : \sigma' \qquad \sigma' \leqslant \sigma}{\Gamma \vdash \mathsf{t} : \sigma}$$

$$\textsc{Gen}\quad \frac{\Gamma \vdash \mathsf{t} : \sigma \qquad \alpha \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \mathsf{t} : \forall\, \alpha.\, \sigma} \qquad \textsc{Fun}\quad \frac{\Gamma, \mathsf{z} : \sigma_2 \vdash \mathsf{t} : \sigma_1}{\Gamma \vdash \lambda \mathsf{z}\,.\, \mathsf{t} : \sigma_2 \to \sigma_1}$$

$$\textsc{App}\quad \frac{\Gamma \vdash \mathsf{t}_1 : \sigma_2 \to \sigma_1 \qquad \Gamma \vdash \mathsf{t}_2 : \sigma_2}{\Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 : \sigma_1}$$

We write $\bar{\alpha}$ for tuples of variables $\alpha_1, ..\alpha_n$ (and more generally $\bar{e}$ for a sequence of elements of the syntactic class e) and $\forall\, \bar{\alpha}.\, \sigma$ for $\forall\, \alpha_1 ... \forall\, \alpha_n.\, \sigma$. We write τ for types that do not contain any universal quantifiers, called monotypes; we write ρ for types that do not have any outermost universal quantifiers. We write $\sigma[\bar{\sigma}/\bar{\alpha}]$ for the simultaneous capture-free substitution of types $\bar{\sigma}$ for variables $\bar{\alpha}$ in σ.

1.2 The generic system $\mathsf{F}(\leqslant)$

Let $\mathsf{F}(\leqslant)$ be the type system for the λ-calculus defined in Figure 1 and parameterized by a binary relation on types $\leqslant$ called *type containment*. A typing context Γ is a finite mapping from program identifiers to types. The empty environment is written $\emptyset$. We write $\Gamma, \mathsf{x} : \sigma$ for the environment that binds x to σ and other program variables as Γ. We write $\mathsf{x} : \sigma \in \Gamma$ to mean that Γ maps x to σ. We write $\mathsf{ftv}(\Gamma)$ for $\bigcup_{\mathsf{x}:\sigma\in\Gamma} \mathsf{ftv}(\sigma)$. Typing judgments are triples of the form $\Gamma \vdash \mathsf{t} : \sigma$. We write $\mathsf{t} \in \mathsf{F}(\leqslant)$ if (and only if) there exists a typing environment Γ and a type σ such that $\Gamma \vdash \mathsf{t} : \sigma$. We may write $\Gamma \vdash_X \mathsf{t} : \sigma$ when we need to remind that typing judgments refer to the type system X.

Rules Var, App, Fun are the same as in the simply typed λ-calculus, except that types may here have quantifiers. Rule Gen allows generalizing the type of an expression over a type variable that does not appear in the environment. The really interesting Rule Inst, which allows replacing the type of an expression with one of its instances, where the notion of instance is determined by the relation $\leqslant$.

System F is obtained by taking $\leq^{\mathsf{F}}$ for $\leqslant$, which is defined as the smallest relation that satisfies the unique axiom Sub of Figure 2. (We use $\leqslant$ to range over arbitrary relations, while symbols $\leq$, $\leq^{\mathsf{F}}$, $\leq^{\eta}$, *etc.* denotes specific relations.) As early as 1984, Mitchell proposed to replace the instance relation $\leq^{\mathsf{F}}$ by a more general relation $\leq^{\eta}$, called *type containment*, which is defined as the smallest relation satisfying the rules of Figure 2 [Mit88]. He also showed that a term t is typable in $\mathsf{F}(\leq^{\eta})$ if and only if there exists a term t' typable in F that is η-equal to t. That is, $\mathsf{F}(\leq^{\eta})$ is the typed closure of F by η-conversion, hence $\mathsf{F}(\leq^{\eta})$ is usually called F^{η}. It follows that the relation $\sigma \leq^{\eta} \sigma'$ holds if and only if there exists a term t of type $\sigma \to \sigma'$ in F that η-reduces to the identity [Mit88]. Such terms are called *coercion* functions.

We write $\equiv$ for the kernel of $\leqslant$. Note that $\leq^{\eta}$ is a structural relation. It is also a subrelation of $\approx$. Conversely, $\leq^{\mathsf{F}}$ is not structural and it is not a subrelation of $\approx$. We write $\leq^{\mathsf{F}}_{\approx}$ for the composition $\leq^{\mathsf{F}} \circ \approx$, which is also equal to $\approx \circ \leq^{\mathsf{F}} \circ \approx$ (but not to $\approx \circ \leq^{\mathsf{F}}$). The relation $\leq^{\mathsf{F}}_{\approx}$ is still not structural, but it is a subrelation of $\approx$. The language $\mathsf{F}(\leq^{\mathsf{F}}_{\approx})$ is not System F *per se*: its typing relation $\vdash_{\leq^{\mathsf{F}}_{\approx}}$ is larger than $\vdash_{\leq^{\mathsf{F}}}$. However, both languages have the same set of typable terms.

Figure 2. Containment rules for $\leq^{\eta}$

$$\textsc{Sub}\quad \frac{\bar{\beta} \notin \mathsf{ftv}(\forall\, \bar{\alpha}.\, \sigma)}{\forall\, \bar{\alpha}.\, \sigma \leqslant \forall\, \bar{\beta}.\, \sigma[\bar{\sigma}/\bar{\alpha}]} \qquad \textsc{Trans}\quad \frac{\sigma \leqslant \sigma' \qquad \sigma' \leqslant \sigma''}{\sigma \leqslant \sigma''}$$

$$\textsc{Arrow}\quad \frac{\sigma_1' \leqslant \sigma_1 \qquad \sigma_2 \leqslant \sigma_2'}{\sigma_1 \to \sigma_2 \leqslant \sigma_1' \to \sigma_2'} \qquad \textsc{All}\quad \frac{\sigma \leqslant \sigma'}{\forall\, \alpha.\, \sigma \leqslant \forall\, \alpha.\, \sigma'}$$

$$\textsc{Distrib}\quad \forall\, \alpha.\, \sigma \to \sigma' \leqslant (\forall\, \alpha.\, \sigma) \to \forall\, \alpha.\, \sigma'$$

About Rule Distrib It is actually possible to limit uses of Rule Distrib to cases where $\alpha \notin \mathsf{ftv}(\sigma)$ and furthermore $\alpha \in \mathsf{ftv}^{-}(\sigma')$ without affecting the type-containment relation $\leq^{\eta}$. We refer to these two limited use of Distrib as Distrib-Right and Distrib-Right-Neg, respectively. Moreover, Rule Distrib-Right is reversible, *i.e.* $\forall\, \alpha.\, \sigma' \leqslant \forall\, \alpha.\, \sigma \to \sigma'$ whenever $\alpha \notin \mathsf{ftv}(\sigma)$. So we actually have $\sigma \to \forall\, \alpha.\, \sigma' \equiv^{\eta} \forall\, \alpha.\, \sigma \to \sigma'$ whenever $\alpha \notin \mathsf{ftv}(\sigma)$. Types can be put in *prenex-form* by repeatedly applying the rewriting rule $\sigma \to \forall\, \alpha.\, \sigma' \rightsquigarrow \forall\, \alpha.\, \sigma \to \sigma'$ when $\alpha \notin \mathsf{ftv}(\sigma)$ in any type context (the side-condition can always be satisfied modulo appropriate renaming). We write $\mathsf{prf}(\sigma)$ for the prenex-form equivalent to σ, which is unique up to $\approx$. We also write $\mathsf{prf}(\Gamma)$ for $\mathsf{prf}(\cdot)$ composed with the mapping Γ. For any judgment $\Gamma \vdash_{\mathsf{F}(\leq^{\eta})} \mathsf{t} : \sigma$, the judgment $\mathsf{prf}(\Gamma) \vdash_{\mathsf{F}(\leq^{\eta})} \mathsf{t} : \mathsf{prf}(\sigma)$ also holds. Moreover, there exists a derivation of this judgment that uses only types in prenex-form in typing judgments. However, subderivations of $\leq^{\eta}$-type-containment judgments may require the use of types not in prenex-form, *e.g.* in applications of Rule Sub.

1.3 Expressiveness of $\mathsf{F}(\leq^{\eta})$

Coercion functions allow the extrusion of quantifiers and deep type-specialization. This is convenient in a type inference context because instantiation may be performed a posteriori. For instance, the term t_K equal to $\lambda \mathsf{z}\,.\, \lambda \mathsf{y}\,.\, \mathsf{y}$ can be assigned either type $\forall\, \alpha.\, \alpha \to \forall\, \beta.\, \beta \to \beta$, giving the sub-expression $\lambda \mathsf{y}\,.\, \mathsf{y}$ a polymorphic type, or type $\forall\, \alpha, \beta.\, \alpha \to \beta \to \beta$. Coercions make both types inter-convertible. Here, either one is actually a principal type of t_K in $\mathsf{F}(\leq^{\eta})$, *i.e.* all other types may be obtained by type-containment [Mit88]. By comparison, there is no type of t_K in System F of which all other types would be $\leq^{\mathsf{F}}$-instances. For this reason, it has been suggested that $\mathsf{F}(\leq^{\eta})$ would be a better candidate for type inference [Mit88]—before full type inference was shown to be undecidable.

The example above shows that *some terms have more types* in $\mathsf{F}(\leq^{\eta})$. One may also expect *more terms to have types* in $\mathsf{F}(\leq^{\eta})$, although we are not aware of any term of $\mathsf{F}(\leq^{\eta})$ that has been proved not to be in F. However, the additional expressive power, if any, does not seem to be very significant.

1.4 Soundness of $\mathsf{F}(\leqslant)$

A typing relation $\Gamma \vdash_X \mathsf{t} : \sigma$ and a reduction relation $\longrightarrow$ on expressions enjoy the subject reduction property if reduction preserves typing. That is, *if* $\Gamma \vdash_X \mathsf{t} : \sigma$ *and* $\mathsf{t} \longrightarrow \mathsf{t}'$, *then* $\Gamma \vdash_X \mathsf{t}' : \sigma$. Of course, this result depends on the particular choice of X and of the reduction relation $\longrightarrow$. In the following, we will consider several type systems $\mathsf{F}(\leqslant)$ where $\leqslant$ is a subrelation of $\leq^\eta$ (or $\leq^{\eta-}$). Type soundness for all of these systems will then follow from type soundness for $\mathsf{F}(\leq^\eta)$.

For an effect-free semantics and in the absence of constants, the relation $\longrightarrow$ is the transitive closure of the one-step reduction $C[(\lambda \mathsf{z}\,.\,\mathsf{t}_1)\ \mathsf{t}_2] \longrightarrow C[\mathsf{t}_1[\mathsf{t}_2/\mathsf{z}]]$ for arbitrary contexts C. Subject reduction is known to hold in System F. It then easily follows from the definition of $\mathsf{F}(\leq^\eta)$ in terms of F that subject reduction also holds in $\mathsf{F}(\leq^\eta)$.

Subject reduction is only halfway key to type soundness. It remains to verify that any well-typed program that is not a value can be further evaluated, that is, the *progress* Lemma. Progress is trivial in the absence of constants. Otherwise, one can show that both subject reduction and progress lemmas hold under some hypotheses ensuring subject reduction and progress for reduction involving constants.

1.5 A stateful sound variant of $\mathsf{F}(\leq^\eta)$

For sake of simplicity, we focus on the treatment of the store and references rather than general side-effecting operations. It is well-known that combining polymorphism and mutable store requires some care. Because System $\mathsf{F}(\leq^\eta)$ is highly polymorphic, we must be even more careful. In this section we define a variant $\mathsf{F}^v(\leq^{\eta-})$ of $\mathsf{F}(\leq^\eta)$ that is safe when equipped with a call-by-value semantics with side effects.

One way to ensure type soundness in the presence of side effects is to give type abstraction a call-by-name semantics. That is, a polymorphic expression must be reevaluated every time it is specialized. In an explicitly typed calculus, polymorphism is introduced by a type abstraction $\Lambda\alpha.\mathsf{t}$, which freezing the evaluation of t and every use of polymorphism is obtained by a type application $\mathsf{t}[\sigma]$, which evaluates the whole expression t (after occurrences of α have been replaced by σ). This call-by-name semantics of polymorphism affects all expressions, whether or not their evaluation may produce side-effects.(Even in the absence of side effects, this modifies sharing of expressions and may change the complexity of computation significantly.) A variant of this solution is to restrict generalization to values, which is known as the value-restriction [Wri95]. More precisely, we may define a class of *non-expansive* expressions that includes at least values and variables, but may also include some other expressions—a formal definition is given below. This solution is now in use in most implementations of ML, although many of them still use a more restrictive definition of non-expansiveness—for no good reason.

We shall thus, as in ML, restrict Rule Gen so that it only applies when the expression t is non-expansive. However, this restriction alone is not sound in $\mathsf{F}(\leq^\eta)$, because it can be bypassed by type-containment. For example, consider the expression t_m defined as $\lambda \mathsf{y}\,.\,\mathsf{t}_r$ where t_r is $\mathsf{ref}\ (\lambda \mathsf{z}\,.\,\mathsf{z})$. As in ML, t_m can be assigned type $\forall\,\alpha.\,\beta \to \mathsf{ref}\ (\alpha \to \alpha)$. Intuitively, this is correct since then an application $\mathsf{t}_m\ \mathsf{t}_0$ will then have the monomorphic type $\mathsf{ref}\ (\alpha \to \alpha)$, or more precisely, will have type $\mathsf{ref}\ (\tau \to \tau)$ for any type τ. The key is that t_m should not have type $\beta \to \forall\,\alpha.\,\mathsf{ref}\ (\alpha \to \alpha)$, since otherwise $\mathsf{t}_m\ \mathsf{t}_0$ would return a reference cell that could be treated as a polymorphic value. This type seems to be disallowed, since the value restriction prohibits generalization of the type of t_r. Unfortunately, type containment allows replacing the correct type $\forall\,\alpha.\,\beta \to \mathsf{ref}\ (\alpha \to \alpha)$ of t_m with the unsound type $\beta \to \forall\,\alpha.\,\mathsf{ref}\ (\alpha \to \alpha)$. The problem can be traced back to rule Distrib-Right-Neg, which allows bypassing the value restriction, and is thus unsound in the presence of side-effects.

The solution is thus both to restrict rule Gen to non-expansive expressions and invalidate Rule Distrib-Right-Neg. Instances or Rule Distrib that are derivable from other rules are both harmless and useful. This is the case in particular when type variable α only occurs positively in $\mathsf{ftv}(\sigma)$. We refer to this special instance of Distrib as Distrib-Right-Pos. It is actually remarkable, that Rule Distrib-Right-Pos, which is valid in $\mathsf{F}^v(\leq^{\eta-})$ leads to the *enhanced* value restriction as defined by Garrigue [Gar04], rather than the standard value-restriction [Wri95].

Store semantics

We assume given a denumerable collection of memory locations $m \in \mathcal{M}$ and restrict constants to the following cases:

$$\begin{array}{rll} \mathsf{c} & ::= \mathsf{c}^+ \mid \mathsf{c}^- & \text{Constants}\\ \mathsf{c}^+ & ::= m & \text{Constructors}\\ \mathsf{c}^- & ::= (\mathsf{ref}\ \cdot) \mid (!\cdot) \mid (\cdot := \cdot) & \text{Destructors} \end{array}$$

Each constant c comes with a fixed arity $a(c)$. The arity of memory locations is 0. The arities of $!\cdot$, $\mathsf{ref}\ \cdot$, and $\cdot := \cdot$ are, respectively 0, 1, and 2, as suggested by the notation.

By lack of space, we cannot describe the call-by-value semantics with store. We refer the reader to an extended version of this paper [Rém05] or to a detailed presentation of type-constraints [PR05].

Typing rules with side effects

Non-expansive expressions $\mathsf{u} \in \mathcal{U}$ are defined as follows:

$$\begin{array}{rll} \mathsf{u} ::= & \mathsf{z} \mid \lambda \mathsf{z}\,.\,\mathsf{t} \mid (\lambda \mathsf{z}_1 \ldots \lambda \mathsf{z}_k\,.\,\mathsf{u})\ \mathsf{u}_1\ \ldots\ \mathsf{u}_k & \\ \mid & \mathsf{c}^+\ \mathsf{u}_1\ \ldots\ \mathsf{u}_k, & k \leq a(\mathsf{c})\\ \mid & \mathsf{c}^-\ \mathsf{u}_1\ \ldots\ \mathsf{u}_k, & k < a(\mathsf{c}) \end{array}$$

By construction, reduction of non-expansive expressions cannot extend the store. Remark that non-expansive expressions may contain free variables. Substituting non-expansive expressions for free variables in non-expansive expressions yields again non-expansive expressions.

We introduce a new *invariant* type symbol ref of arity 1 to classify expressions that evaluate to store locations. Invariance mean that an occurrence of a variable in a type $\mathsf{ref}\ \sigma$ is both positive and negative and thus never in $\mathsf{ftv}^\dagger(\sigma)$. Moreover, a structural relation $\leqslant$ must now validate the following rule:

$$\textsc{Ref}\quad \frac{\sigma \leqslant \sigma' \qquad \sigma' \leqslant \sigma}{\mathsf{ref}\ \sigma \leqslant \mathsf{ref}\ \sigma'}$$

We write $\leq^{\eta-}$ for the smallest relation $\leqslant$ that satisfies rules Sub, Trans, Arrow, All of Figure 2 and rule Ref.

We also restrict generalization to non-expansive expressions *or* to non-negative type variables.

$$\textsc{Gen}^v\quad \frac{\Gamma \vdash \mathsf{t} : \sigma \qquad \alpha \notin \mathsf{ftv}(\Gamma) \qquad \mathsf{t} \in \mathcal{U}\ \vee\ \alpha \in \mathsf{ftv}^\dagger(\sigma)}{\Gamma \vdash \mathsf{t} : \forall\,\alpha.\,\sigma}$$

Let $\mathsf{F}^v(\leqslant)$ be the type system $\mathsf{F}(\leqslant)$ where Rule Gen has been replaced by Rule Gen^v. Types for constants are defined by an initial environment Γ_0 that contains exactly:

$$\begin{array}{rl} \mathsf{ref}\ \cdot : & \forall\,\alpha.\,\alpha \to \mathsf{ref}\ \alpha\\ !\cdot : & \forall\,\alpha.\mathsf{ref}\ \alpha \to \alpha\\ \cdot := \cdot : & \forall\,\alpha.\mathsf{ref}\ \alpha \to \alpha \to \alpha \end{array}$$

With these restrictions, we still have $\Gamma_0 \vdash_{\mathsf{F}^v(\leq^{\eta-})} \mathsf{t}_m : \forall\,\alpha.\,\mathsf{unit} \to \mathsf{ref}\ (\alpha \to \alpha)$, but not $\Gamma_0 \vdash_{\mathsf{F}^v(\leq^{\eta-})} \mathsf{t}_m : \mathsf{unit} \to \forall\,\alpha.\,\mathsf{ref}\ (\alpha \to \alpha)$ any longer.

THEOREM 1. *The language $\mathsf{F}^v(\leq^{\eta-})$ with references is sound.*

By lack of space, we refer to [Rém05] for a more formal statement of this result and its proof. Type soundness is shown in a standard way by combination of subject reduction and progress lemmas.

1.6 Predicative fragments $\mathsf{F}(\leq_p^{\mathsf{F}})$ and $\mathsf{F}(\leq_p^{\eta})$

The predicative fragment of System F is the system $\mathsf{F}(\leq_p^{\mathsf{F}})$ obtained by restricting the instance relation $\leq^{\mathsf{F}}$ so that only monotypes can be substituted for type variables[1]. That is, $\leq_p^{\mathsf{F}}$ is the smallest relation satisfying the following rule:

$$\textsc{Sub}_p\quad \frac{\bar\beta \notin \mathsf{ftv}(\forall\,\bar\alpha.\,\sigma)}{\forall\,\bar\alpha.\,\sigma \leqslant \forall\,\bar\beta.\,\sigma[\bar\tau/\bar\alpha]}$$

The predicative fragment of $\mathsf{F}(\leq^{\eta})$ can be defined either (1) as the typed-closure of $\mathsf{F}(\leq_p^{\mathsf{F}})$ by η-conversion, or (2) as $\mathsf{F}(\leq_p^{\eta})$ where $\leq_p^{\eta}$ is the smallest relation satisfying all rules of Figure 2 except Rule SUB, which is replaced by Rule SUB$_p$. Fortunately, definitions (1) and (2) are equivalent.

The type system $\mathsf{F}(\leq_p^{\eta})$ is safe, as it is a subset of $\mathsf{F}(\leq^{\eta})$. This does not imply subject reduction. However, one may show independently that subject reduction holds in both $\mathsf{F}(\leq_p^{\mathsf{F}})$ and in $\mathsf{F}(\leq_p^{\eta})$.

Replacing DISTRIB by DISTRIB-RIGHT in the definition of $\leq_p^{\eta}$ does not change the relation itself. This is not a consequence of the similar result for the impredicative relation $\leq^{\eta}$. Indeed, further replacing DISTRIB-RIGHT by DISTRIB-RIGHT-NEG would lead to a weaker relation than $\leq_p^{\eta}$. Hence, in the case of a call-by-value stateful semantics the language $\mathsf{F}^v(\leq_p^{\eta-})$ could be safely enriched with a version of DISTRIB that requires α in $\mathsf{ftv}^\dagger(\sigma') \setminus \mathsf{ftv}(\sigma)$. We refer to this Rule as DISTRIB-RIGHT-POS.

1.7 Expressiveness of the predicative fragments

The restriction of Systems F or $\mathsf{F}(\leq^{\eta})$ to their predicative fragments comes with a significant loss of expressiveness. It means that polymorphism is limited to simple types. That is, programs can manipulate values of several types that differ in their *monomorphic* parts but must have the same *polymorphic* shape. For instance, `int` $\to$ `int` and `bool` $\to$ `bool` can be two specializations of a predicative type, but $(\forall\,\alpha.\,\alpha \to \alpha) \to (\forall\,\alpha.\,\alpha \to \alpha)$ and `int` $\to$ `int` cannot.

The restriction of rule SUB to rule SUB$_p$ suggests a large family of counter-examples. For instance, a polymorphic function of type $\forall\,\alpha.\,\sigma \to \sigma'$ can only be used at types $\sigma[\tau/\alpha]$ where τ is a monotype. Thus, one may need as many copies of the function as there are different shapes σ at which the function needs to be used. As a particular case, the apply function $\lambda\mathsf{f}\,.\,\lambda\mathsf{z}\,.\,\mathsf{f\ z}$ does not have a most general type, but as many types as there are polymorphic shapes for the type of the function f. This is a general situation that also applies to iterators such as `iter`, `map`, `fold`, *etc.*

Unsurprisingly, the well-known encoding of existential types with universal types in System F is also severely limited in the predicative fragments. A value v of an existential type $\exists\alpha.\sigma$ can be encoded as a function $\forall\,\beta.\,(\forall\,\alpha.\,\sigma \to \beta) \to \beta$.

[1] One may also consider a stratification of predicative fragments—see [Lei91]

Figure 3. Type containment rules for $\leq_p^{\eta-}$

$$\textsc{Refl}\quad \alpha \leqslant \alpha \qquad\qquad \textsc{Arrow}\quad \frac{\sigma_1' \leqslant \sigma_1 \qquad \sigma_2 \leqslant \sigma_2'}{\sigma_1 \to \sigma_2 \leqslant \sigma_1' \to \sigma_2'}$$

$$\textsc{All-I}\quad \frac{\sigma \leqslant \sigma' \qquad \alpha \notin \mathsf{ftv}(\sigma)}{\sigma \leqslant \forall\,\alpha.\,\sigma'} \qquad\qquad \textsc{All-E}\quad \frac{\sigma[\tau/\alpha] \leqslant \rho}{\forall\,\alpha.\,\sigma \leqslant \rho}$$

The use of value v under the name x inside some term t is encoded as the application of v to $\lambda x\,.\,\mathsf{t}$. This application can be typed in F, giving x the polymorphic type $\forall\,\alpha.\,\sigma \to \sigma'$ and instantiating β with σ'. In the predicative fragment, however, σ' must be a monotype. Hence, the restriction of existential types to the predicative fragment only allows expressions that manipulate (encoded) existential types to return monotypes. Moreover, the type hidden by the abstraction may only be a monotype. In explicit System F, the expression $\mathsf{pack\ t\ as\ z} : \exists\,\beta.\,\sigma$, where $\sigma[\sigma'/\beta]$ is the type of t, is encoded as $\Lambda\alpha.\,\lambda\mathsf{f} : \forall\,\beta.\,\sigma\,.\,\mathsf{f}\ \sigma'\ \mathsf{t}$ where σ' is the abstract part of the type. Hence, σ' must be a monotype τ in $\mathsf{F}(\leq_p^{\mathsf{F}})$. As a corollary, encodings of objects [BCP99] do not work well in the predicative fragments: the object state cannot be hidden any longer as soon as it references an object.

Indeed, we may exhibit terms than are typable in F but not in the predicative fragment $\mathsf{F}(\leq_p^{\mathsf{F}})$. One such term $(\lambda\mathsf{z}\,.\,\mathsf{z\ y\ z})\ (\lambda\mathsf{z}\,.\,\mathsf{z\ y\ z})$ is due to Pawel Urzyczyn, according to Leivant [Lei91]. Stratified versions of System F are more expressive than System $\mathsf{F}(\leq_p^{\mathsf{F}})$ [Lei91]. However, they would not ease type inference.

2. Type inference in the language $\mathsf{F_{ML}}$

Full type inference is undecidable in both System F [Wel94] and $\mathsf{F}(\leq^{\eta})$ [Wel96]—the type-containment relation $\leq^{\eta}$ is itself undecidable [TU02, Wel95]. To the best of our knowledge it is not known whether type inference is decidable for the predicative fragments. However, Harper and Pfenning remarked that the reduction of second-order unification to type inference for System F with explicit placeholders for type abstractions and applications also applies to the predicative fragment [Pfe88]. Still, we do not know whether full type inference for $\mathsf{F}(\leq_p^{\eta})$ is decidable, even though predicative type containment $\leq_p^{\eta}$ is itself decidable [PVWS05b].

We also consider the restriction $\leq_p^{\eta-}$ of $\leq_p^{\eta}$ obtained by removing Rule DISTRIB from the definition of $\leq_p^{\eta}$. That is, we define $\leq_p^{\eta-}$ as the smallest relation that satisfies rules SUB$_p$, TRANS, ARROW and ALL. A consequence of the removal of rule DISTRIB is that $\leq_p^{\eta-}$ has a very simple syntax-directed presentation given by rules of Figure 3.

LEMMA 1. *$\leq_p^{\eta-}$ is also the smallest relation $\leqslant$ that satisfies the rules of Figure 3.*

LEMMA 2 (Stability of $\leq_p^{\eta-}$ by substitution). *If $\sigma \leq_p^{\eta-} \sigma'$ then $\sigma[\bar\tau/\bar\alpha] \leq_p^{\eta-} \sigma'[\bar\tau/\bar\alpha]$.*

Actually, the two relations $\leq_p^{\eta}$ and $\leq_p^{\eta-}$ coincide on types in prenex form, as shown by the following lemma due to Peyton-Jones, Vytiniotis, Weirich and Shields [PVWS05b].

LEMMA 3 (Peyton-Jones *et al.*). *$\sigma \leq_p^{\eta} \sigma'$ if and only if $\mathsf{prf}(\sigma) \leq_p^{\eta-} \mathsf{prf}(\sigma')$.*

This provides us with an algorithm for testing $\leq_p^\eta$ by first projecting both types to their prenex forms and then testing for $\leq_p^{\eta-}$ (or by deriving a direct algorithm from this composition [PVWS05b]. Although the restriction $\leq_p^{\eta-}$ of $\leq_p^\eta$ is primarily introduced to ensure soundness in $\mathsf{F}^v(\leq_p^{\eta-})$, we may also consider the language $\mathsf{F}(\leq_p^{\eta-})$ in the absence of side-effects. One advantage is that type inference for $\mathsf{F}(\leq_p^{\eta-})$ is closer to type inference for ML, especially in its treatment of generalization. Moreover, working with $\leq_p^{\eta-}$ is slightly more convenient than $\leq_p^\eta$ for type inference. We start with this simpler case and will consider $\mathsf{F}(\leq_p^\eta)$ and $\mathsf{F}^v(\leq_p^{\eta-})$ in sections 2.5 and 2.6.

2.1 Terms with type annotations: $\mathsf{F}_{\mathsf{ML}}(\leqslant)$

We leave the relation $\leqslant$ a parameter of $\mathsf{F}_{\mathsf{ML}}(\leqslant)$ so as to share some of the developments between $\leq_p^{\eta-}$ and $\mathsf{F}_{\mathsf{ML}}(\leq_p^{\eta})$. However, the intention is that the type-containment relation $\leqslant$ be predicative, *i.e.* a subrelation of $\mathsf{F}_{\mathsf{ML}}(\leq_p^{\eta})$.

Thus, the type-containment relations $\leqslant$ that we shall consider are decidable, and so is $\leqslant$-constraint resolution, as we shall see below. However, this is not sufficient to perform type inference. In order to avoid *guessing* polymorphic types, we now extend terms with type annotations. An annotation is a type scheme σ whose free variables $\bar\alpha$ are locally bound; we write $\exists\,\bar\alpha.\,\sigma$. While σ represents the explicit (usually second-order) type information, the existentially bound variables $\bar\alpha$ represent the implicit type information, which will be inferred. That is, type annotations are partial, which is important to leave room for inference. As a particular case, the trivial annotations $\exists\,\alpha.\,\alpha$ let us infer monomorphic types, as in ML. We require that type annotations be closed. This is only for the sake of simplicity. One could easily allow type annotations to have free type variables, provided these are explicitly bound somewhere in the program. This mechanism, sometimes known as *scoped type variables*, is discussed by Peyton Jones and Shields [PS03] and by Pottier and Rémy [PR]. We let θ range over type annotations.

The syntax of $\mathsf{F}_{\mathsf{ML}}(\leqslant)$ is as follows:

$$\mathsf{t} ::= \mathsf{x} \mid \lambda\mathsf{z}\,.\,\mathsf{t} \mid \mathsf{t}_1\ (\mathsf{t}_2 : \theta) \mid \mathsf{let}\ \mathsf{z} = (\mathsf{t}_1 : \theta)\ \mathsf{in}\ \mathsf{t}_2$$

As usual, expressions may be identifiers x, abstractions $\lambda\mathsf{z}\,.\,\mathsf{t}$, or applications $\mathsf{t}_1\ (\mathsf{t}_2 : \theta)$. However, arguments of applications must now always be explicitly annotated (although annotations may be trivial). The intuition for annotating the argument of applications is to avoid guessing the type of the argument. Indeed, knowing the type of the result of an application does not tell anything about the type of its argument. We also allow let-bindings $\mathsf{let}\ \mathsf{z} = (\mathsf{t}_1 : \theta)\ \mathsf{in}\ \mathsf{t}_2$, which mean $(\lambda\mathsf{z}\,.\,\mathsf{t}_2)\ (\mathsf{t}_1 : \theta)$, but which will be typed in a special way, as in ML. As for applications, an annotation is required for the expected type of t_1, since it cannot be deduced from the expected type of the whole expression.

2.2 Typing rules

Typing rules for $\mathsf{F}_{\mathsf{ML}}(\leqslant)$ are described in Figure 4. They use judgments of the form $\Gamma \vdash \mathsf{t} : \sigma$ as for $\mathsf{F}(\leqslant)$. These judgments should be read as checking judgments, where Γ, t and σ are given. Rules Var, Inst, Gen, and Fun are all taken from $\mathsf{F}(\leqslant)$. However, as mentioned above, the intention is that $\leqslant$ in the right premise of Rule Inst be chosen as an instance of $\leq_p^\eta$, so as to ensure predicativity. That is, the polymorphic structure of polymorphic types can only be instantiated by monotypes. Note that, as opposed to ML, rule Fun allows its argument to be polymorphic. Despite the appearance, this does not imply guessing polymorphism, as long as the expected type $\sigma_2 \to \sigma_1$ is given.

Figure 4. Typing rules for $\mathsf{F}_{\mathsf{ML}}(\leqslant)$

Var Inst Gen Fun

App$_\mathsf{A}$
$$\frac{\Gamma \vdash \mathsf{t}_1 : \sigma_2[\bar\tau/\bar\beta] \to \sigma_1 \qquad \Gamma \vdash \mathsf{t}_2 : \sigma_2[\bar\tau/\bar\beta]}{\Gamma \vdash \mathsf{t}_1\ (\mathsf{t}_2 : \exists\,\bar\beta.\,\sigma_2) : \sigma_1}$$

Let$_\mathsf{A}$
$$\frac{\Gamma \vdash \mathsf{t}_1 : \forall\,\bar\alpha.\,\sigma_1[\bar\tau/\bar\beta] \qquad \Gamma, \mathsf{z} : \forall\,\bar\alpha.\,\sigma_1[\bar\tau/\bar\beta] \vdash \mathsf{t}_2 : \sigma_2}{\Gamma \vdash \mathsf{let}\ \mathsf{z} = (\mathsf{t}_1 : \exists\,\bar\beta.\,\sigma_1)\ \mathsf{in}\ \mathsf{t}_2 : \sigma_2}$$

Rule App$_\mathsf{A}$ (the subscript is to remind the *annotation*) is not quite the one of $\mathsf{F}(\leqslant)$. We must of course take the annotation of the argument into account: the type of t_2, which is also the domain of the type of t_1, must be an instance of the annotation. The annotation avoids guessing the polymorphic parts of the expected type of t_1 and t_2 that cannot be deduced from the expected type of the application. When the annotation is the trivial one, $\sigma_2[\bar\tau/\bar\alpha]$ must be a monotype τ_2, to be guessed—not a problem, since it is a monotype. More generally, only monotypes $\bar\tau$ need to be guessed in applications. Rule Let$_\mathsf{A}$ is taken from ML, except that the annotation on t_1 is used to build the expected type of t_1, up to first-order instantiation, much as in Rule App$_\mathsf{A}$.

Expressiveness. Apart from annotations, the language $\mathsf{F}_{\mathsf{ML}}(\leqslant)$ is as expressive as $\mathsf{F}(\leqslant)$ for any relation $\leqslant$. Formally, let the type erasure of a term t be the λ-term obtained by dropping all type annotations and replacing all let-bindings $\mathsf{let}\ \mathsf{z} = \mathsf{t}_1\ \mathsf{in}\ \mathsf{t}_2$ by $(\lambda\mathsf{z}\,.\,\mathsf{t}_2)\ \mathsf{t}_1$. Then, we have $\mathsf{t} \in \mathsf{F}(\leqslant)$ if and only if there exists an expression $\mathsf{t}' \in \mathsf{F}_{\mathsf{ML}}(\leqslant)$ whose erasure is t.

Syntactic sugar. While annotations are always required in applications and let bindings, the *trivial annotations* may be used. This is not exactly an absence of annotation, since the annotation is mandatory and a trivial annotation always stands for a monomorphic type. In particular, an expression where all annotations are trivial will be typed as in ML. For convenience, we may allow $\mathsf{let}\ \mathsf{z} = \mathsf{t}_1\ \mathsf{in}\ \mathsf{t}_2$ and $\mathsf{t}_1\ \mathsf{t}_2$ as syntactic sugar for $\mathsf{let}\ \mathsf{z} = (\mathsf{t}_1 : \exists\,\alpha.\,\alpha)\ \mathsf{in}\ \mathsf{t}_2$ and $\mathsf{t}_1\ (\mathsf{t}_2 : \exists\,\alpha.\,\alpha)$, respectively.

The language $\mathsf{F}_{\mathsf{ML}}(\leqslant)$ is a conservative extension to ML. That is, if Γ is an ML environment and t is an ML expression, *i.e.* both without any non trivial annotation, then $\Gamma \vdash_{\mathsf{ML}} \mathsf{t} : \tau$ if and only if $\Gamma \vdash_{\mathsf{F}_{\mathsf{ML}}(\leqslant)} \mathsf{t} : \tau$.

The language does not allow arbitrary expressions to carry type annotations. However, we can define $(\mathsf{t} : \theta)$ as syntactic sugar for $\mathsf{let}\ \mathsf{z} = (\mathsf{t} : \theta)\ \mathsf{in}\ \mathsf{z}$.

Example. The expression $\lambda\mathsf{z}\,.\,\mathsf{z}$ is well-typed, and can be given type $\tau \to \tau$, for any type τ, as in ML. It may also be given types $\forall\,\alpha.\,\alpha \to \alpha$, $(\forall\,\alpha.\,\alpha) \to (\forall\,\alpha.\,\alpha)$, $\forall\,\beta.((\forall\,\alpha.\,\alpha) \to \beta) \to ((\forall\,\alpha.\,\alpha) \to \beta)$, *etc.* none of which is more general than the other in $\mathsf{F}_{\mathsf{ML}}(\leq_p^{\eta-})$. In $\mathsf{F}(\leq_p^\eta)$, the first one is better than the second-one; (In $\mathsf{F}(\leq^\eta)$, the first-one is better than all other ones on this particular example, but this is accidental.) Hence $\mathsf{F}_{\mathsf{ML}}(\leq_p^{\eta-})$ does not have principal types in the usual sense. However, as we shall see below, it has a principal type for any given polymorphic *shape*.

Figure 5. Syntax-directed typing rules $\mathsf{F_A}$ for $\mathsf{F_{ML}}(\leq_p^{\eta-})$

Var-Inst-Rho
$$\frac{\mathsf{x} : \sigma \in \Gamma \qquad \sigma \leq_p^{\eta-} \rho}{\Gamma \vdash \mathsf{x} : \rho} \qquad \textsc{Gen} \qquad \textsc{Fun}$$

App$_\mathsf{A}$-Rho
$$\frac{\Gamma \vdash \mathsf{t}_1 : \sigma_2[\bar\tau/\bar\beta] \to \rho_1 \qquad \Gamma \vdash \mathsf{t}_2 : \sigma_2[\bar\tau/\bar\beta]}{\Gamma \vdash \mathsf{t}_1\ (\mathsf{t}_2 : \exists\,\bar\beta.\,\sigma_2) : \rho_1}$$

Let$_\mathsf{A}$-Gen-Rho
$$\frac{\Gamma \vdash \mathsf{t}_1 : \sigma_1[\bar\tau/\bar\beta] \qquad \Gamma, \mathsf{z} : \forall\backslash\Gamma.\,\sigma_1[\bar\tau/\bar\beta] \vdash \mathsf{t}_2 : \rho_2}{\Gamma \vdash \mathsf{let}\ \mathsf{z} = (\mathsf{t}_1 : \exists\,\bar\beta.\,\sigma_1)\ \mathsf{in}\ \mathsf{t}_2 : \rho_2}$$

As in ML, the expression $\lambda\mathsf{z}\,.\,\mathsf{z}\ \mathsf{z}$ does not have any monotype τ. However, we may explicitly provide the information that we expect a type of the form $(\forall\,\alpha.\,\alpha \to \alpha) \to \tau \to \tau$ for some τ. Then, the expression is well-typed (and a best type of *that shape* may be inferred). We have $\Gamma \vdash_{\mathsf{F_{ML}}(\leq_p^{\eta-})} \lambda\mathsf{z}\,.\,\mathsf{z}\ \mathsf{z} : \forall\,\beta.(\forall\,\alpha.\,\alpha \to \alpha) \to \beta \to \beta$.

Typing problems. Let φ range over first-order substitutions, simply called substitutions for short, which map type variables to monotypes. We may distinguish four different problems:

1. *Typability:* Given a term t, do there exist a context Γ and a type σ such that $\Gamma \vdash \mathsf{t} : \sigma$?
2. *Typing inference:* Given a term t, what are the pairs of a context Γ and a type σ such that $\Gamma \vdash \mathsf{t} : \sigma$?
3. *Type inference:* Given a term t and a context Γ, what are the types σ and substitutions φ such that $\Gamma\varphi \vdash \mathsf{t} : \sigma$?
4. *Type checking:* Given a term t, a context Γ, and a type σ, what are the substitutions φ such that $\Gamma\varphi \vdash \mathsf{t} : \sigma\varphi$?

Note that our definition of typechecking still allows first-order inference, which is somehow non standard. Neither typing inference, nor type inference problems have principal solutions in general in $\mathsf{F_{ML}}(\leq_p^{\eta})$, nor in $\mathsf{F_{ML}}(\leq_p^{\eta-})$. However, type checking problems do.

2.3 Syntax-directed typing rules $\mathsf{F_A}$

As usual, typing judgments need not use all the flexibility allowed by the typing rules. That is, typing derivations can be rearranged to form derivations that follow certain patterns. In particular, derivations that are driven by the syntax of the conclusion, called syntax-directed, are the key to type checking and type inference. Figure 5 presents an equivalent set of rules for deriving typing judgments in $\mathsf{F_{ML}}(\leq_p^{\eta-})$. We write $\forall\backslash\Gamma.\,\sigma$ for the type $\forall\,(\mathsf{ftv}(\sigma) \setminus \mathsf{ftv}(\Gamma)).\,\sigma$.

Rules Gen and Fun are unchanged. All syntax-directed rules but Gen assign ρ-types rather that types to terms. As opposed to ML, the Gen rule is not removed in the syntax-directed system. It is the only rule that may (and thus must) be used first to derive judgments whose conclusion mentions a type scheme that is not a ρ-type. Hence, it can be used either as the last rule in a derivation, or in three other places (and only there): immediately above the left-premise of a Let rule, the premise of a Fun rule or the premise of another Gen. Strictly speaking, Rule Gen is not syntax-directed, unless we take the expected type-scheme as part of the syntax. It would be possible and obvious to inline it in the premises of both rules Fun and Let, but this would be more verbose and less readable. Rule Var-Inst-Rho behaves as Var followed by Inst; Rule App$_\mathsf{A}$-Rho is a restriction of Rule App$_\mathsf{A}$ so as to conclude ρ-types only (hence the -Rho suffix). Rule Let$_\mathsf{A}$-Gen-Rho behaves as Let (restricted to ρ-types) with a sequence of Gen above its left premise. As in ML, this is the most interesting rule. The type scheme required for t_1 is the one requested by the annotation, but where monomorphic parts may be instantiated. Hence, its free types variables that do not appear in Γ can be generalized in the type assigned to z while typechecking t_2. This is the only place where we introduce polymorphism that is not explicitly given (in $\mathsf{F_A}$, Rule Gen may only be used for polymorphism that is explicitly given). We do so, much as in ML, and without really guessing polymorphism—we just have it for free! Shapes of all other polymorphic types in the premises are never inferred and just taken from some corresponding type in the conclusion.

If we change all type annotations into $\exists\,\beta.\,\beta$ in syntax-directed rules, we obtain exactly the syntax-directed rules of ML (Gen becomes useless if the expected type is a simple type). Thus, we depart from ML only by allowing second-order polymorphic types in annotations and typing judgments. The instantiation, let-polymorphism, and inference mechanism remain the same. In particular, type inference (as defined above) relies on first order-unification, which we shall see now.

First, let us state a series of useful standard Lemmas.

Lemma 4 (Substitution). *If $\Gamma \vdash_{\mathsf{F_A}} \mathsf{t} : \sigma$, then there exists a derivation of $\Gamma\varphi \vdash_{\mathsf{F_A}} \mathsf{t} : \sigma\varphi$ of the same length for any substitution φ.*

We write $\Gamma' \leq_p^{\eta-} \Gamma$ if $dom(\Gamma) = dom(\Gamma')$ and $\Gamma'(\mathsf{z}) \leq_p^{\eta-} \Gamma(\mathsf{z})$ for each z in $dom(\Gamma)$,

Lemma 5 (Strengthening). *If $\Gamma \vdash_{\mathsf{F_A}} \mathsf{t} : \sigma$ and $\Gamma' \leq_p^{\eta-} \Gamma$, then $\Gamma' \vdash_{\mathsf{F_A}} \mathsf{t} : \sigma$.*

The next two lemmas establish the correspondence between the two presentations of the typing rules.

Lemma 6 (Soundness of syntax-directed rules). *Rules* Var-Inst-Rho, App$_\mathsf{A}$-Rho, *and* Let$_\mathsf{A}$-Gen-Rho *are derivable in $\mathsf{F_{ML}}(\leq_p^{\eta-})$,* (i.e. *they may be replaced by chunk of derivations using rules of $\mathsf{F_{ML}}(\leq_p^{\eta-})$*).

Lemma 7. *(Completeness of syntax-directed rules) Rules* Inst, App$_\mathsf{A}$, *and* Let$_\mathsf{A}$ *are admissible in $\mathsf{F_A}$,* (i.e. *adding them to the rules to the definition of $\mathsf{F_A}$ does not allow to derive more judgments).*

This direction is more interesting and a bit trickier than the previous one. There are a few subtleties. Admissibility of Rule Inst would not be true if $\leq_p^{\eta-}$ were replaced by $\leq_p^{\eta}$ in both systems. Admissibility of Rule App$_\mathsf{A}$ requires to follow an application of Rule App$_\mathsf{A}$-Rho with a sequence of instances of Rule Gen. This would not be true if Gen were replaced by Gen^v in both systems—see Rule App$_\mathsf{A}{}^v$ in Figure 9.

Corollary 8. *The syntax-directed rules and the original rules derive the same judgments.*

2.4 Type inference via type constraints

Syntax-directed typing rules open the path to a type inference algorithm. They show that a typing judgment for an expression can only hold if some judgment for its immediate subexpressions also hold. Moreover, the type judgments for sub-expressions are entirely determined from the original

Figure 6. Syntax of constraints

$$\begin{aligned} C ::= {} & \mathsf{true} \mid \mathsf{false} \mid \tau = \tau \mid \sigma \leq \sigma \\ & \mid C \wedge C \mid \exists\,\alpha.\,C \mid \forall\,\alpha.\,C \\ & \mid \mathsf{x} \preceq \sigma \mid \mathsf{def\ x} : \forall\bar{\alpha}[C].\sigma \ \mathsf{in}\ C \end{aligned}$$

Abbreviations:

$$\begin{aligned} (\forall\bar{\alpha}[C].\sigma) \preceq \rho &\triangleq \exists\,\bar{\alpha}.(C \wedge \sigma \leq \rho) \qquad \bar{\alpha} \notin \mathsf{ftv}(\rho) \\ \mathsf{let\ x} : \forall\bar{\alpha}[C].\sigma\ \mathsf{in}\ C' &\triangleq (\exists\,\bar{\alpha}.\,C) \wedge (\mathsf{def\ x} : \forall\bar{\alpha}[C].\sigma\ \mathsf{in}\ C') \\ \mathsf{let}\ \Gamma, \mathsf{x} : \forall\bar{\alpha}[C].\sigma\ \mathsf{in}\ C' &\triangleq \mathsf{let}\ \Gamma\ \mathsf{in\ let\ x} : \forall\bar{\alpha}[C].\sigma\ \mathsf{in}\ C' \\ \mathsf{let}\ \emptyset\ \mathsf{in}\ C &\triangleq C \end{aligned}$$

judgment except for some instantiations of type variables by monotypes. This is just as in ML and suggests an underlying first-order type-inference mechanism.

Typing constraints are a general yet simple and intuitive framework to define type inference algorithms for ML-like languages [PR05]. Below, we present a brief summary of this approach and extend it in a straightforward manner to cover type inference for $\mathsf{F}_{\mathsf{ML}}(\leqslant)$. We refer the reader to [PR05] for a more thorough presentation.

The syntax of type constraints is given in Figure 6. We use letter C to range over type constraints. Default atomic constraints are true, false, equality constraints $\tau = \tau$, and subtyping constraints $\sigma \leq \sigma'$. We build constraints by conjunction $C \wedge C'$, existential quantification $\exists\,\alpha.\,C$, or universal quantification $\forall\,\alpha.\,C$. Finally, we use two special forms of constraints for polymorphism elimination $\mathsf{x} \preceq \rho$ and polymorphism introduction $\mathsf{def\ x} : \forall\bar{\alpha}[C].\rho\ \mathsf{in}\ C'$.

In order to interpret constraints, we need a model in which we may interpret free type variables. As for ML, we take the set of closed ground types (monotypes without type variables) for our model. (Here, we must assume at least one type constructor of arity 0, *e.g.* int.) We use letter t to range over ground types. A ground assignment ϕ is a map from all type variables to elements of the model. We also see ϕ as a mapping from types to types, by letting $(\sigma_1 \to \sigma_2)\phi = \sigma_1\phi \to \sigma_2\phi$ and $(\forall\,\alpha.\,\sigma)\phi = \forall\,\alpha.(\sigma\phi_\alpha)$ where ϕ_α is the restriction of ϕ to $dom(\phi) \setminus \{\alpha\}$. (Consistently with the notation for substitutions, we write $\sigma\phi$ for the application of ϕ to σ.)

We write $\phi \Vdash C$ to mean that C is valid in environment ϕ. We take $\phi \Vdash \mathsf{true}$ and $\phi \nVdash \mathsf{false}$ for any ϕ. We take $\phi \Vdash \tau = \tau'$ if and only if $\tau\phi \leq_p^{\eta-} \tau'\phi$ and We take $\phi \Vdash \sigma \leq \sigma'$ if and only if $\sigma\phi \leq_p^{\eta-} \sigma'\phi$. Technically, we could identify the equation $\tau = \tau'$ with the inequation $\tau \leq \tau'$, since they coincide. However, for sake of exposition, we prefer to explicitly transform subtyping constraints into equations when both sides are monotypes. We take $\phi \Vdash C \wedge C'$ if and only if $\phi \Vdash C$ and $\phi \Vdash C'$. We take $\phi \Vdash \exists\,\alpha.\,C$ if and only if there exists a ground type t such that $\phi[\alpha \mapsto t] \Vdash C$ (we write $\phi[\alpha \mapsto t]$ for the environment that maps α to t and otherwise coincides with ϕ). We take $\phi \Vdash \forall\,\alpha.\,C$ if and only if for every ground type t, we have $\phi[\alpha \mapsto t] \Vdash C$. A constraint C entails a constraint C' if for every ground assignment ϕ, such that $\phi \Vdash C$ we also have $\phi \Vdash C'$. We then write $C \Vdash C'$. Contraints C and C' are equivalent if both $C \Vdash C'$ and $C' \Vdash C$ holds.

Def-constraints $\mathsf{def\ x} : \forall\bar{\alpha}[C].\sigma\ \mathsf{in}\ C'$ can be interpreted as $C'[\forall\bar{\alpha}[C].\sigma/\mathsf{x}]$ (of course, the resolution of constraints will avoid this substitution). The effect of this substitution is to expose constraints of the form $\forall\bar{\alpha}[C].\sigma \preceq \rho$. These are syntactic sugar for $\exists\,\bar{\alpha}.(C \wedge \sigma \leq \rho)$, provided $\bar{\alpha} \notin \mathsf{ftv}(\rho)$. Hence, the scope in a constrained type scheme $\forall\bar{\alpha}[C].\sigma$ is both C and σ. In fact, rather that using $\mathsf{def\ x} : \forall\bar{\alpha}[C].\rho\ \mathsf{in}\ C'$ directly, we often use $\mathsf{let\ x} : \forall\bar{\alpha}[C].\rho\ \mathsf{in}\ C'$ as an abbreviation for $\exists\,\bar{\alpha}.\,C \wedge \mathsf{def\ x} : \forall\bar{\alpha}[C].\rho\ \mathsf{in}\ C'$.

Figure 7. Solving $\leq$-constraints

$$\begin{aligned} \tau \leq \tau' &\longrightarrow \tau = \tau' \\ \sigma_1 \to \sigma_2 \leq \alpha &\longrightarrow \exists\,\alpha_1\alpha_2.(\sigma_1 \to \sigma_2 \leq \alpha_1 \to \alpha_2 \\ &\qquad\qquad \wedge\ \alpha = \alpha_1 \to \alpha_2) \\ \alpha \leq \sigma_1 \to \sigma_2 &\longrightarrow \exists\,\alpha_1\alpha_2.(\alpha_1 \to \alpha_2 \leq \sigma_1 \to \sigma_2 \\ &\qquad\qquad \wedge\ \alpha = \alpha_1 \to \alpha_2) \\ \sigma_1 \to \sigma_2 \leq \sigma_1' \to \sigma_2' &\longrightarrow \sigma_1' \leq \sigma_1 \wedge \sigma_2 \leq \sigma_2' \\ \forall\,\alpha.\,\sigma \leq \rho &\longrightarrow \exists\,\alpha.(\sigma \leq \rho) \qquad \alpha \notin \mathsf{ftv}(\rho) \\ \sigma \leq \forall\,\alpha.\,\sigma' &\longrightarrow \forall\,\alpha.(\sigma \leq \sigma') \qquad \alpha \notin \mathsf{ftv}(\sigma) \end{aligned}$$

Figure 8. Constraint generation rules

$$\begin{aligned} [\![\mathsf{x} : \rho]\!] &= \mathsf{x} \preceq \rho \\ [\![\lambda \mathsf{z}\,.\,\mathsf{t} : \alpha]\!] &= \exists\,\beta_2\beta_1.([\![\lambda \mathsf{z}\,.\,\mathsf{t} : \beta_2 \to \beta_1]\!] \\ &\qquad \wedge\ \alpha = \beta_2 \to \beta_1) \qquad \beta_1, \beta_2 \neq \alpha \\ [\![\lambda \mathsf{z}\,.\,\mathsf{t} : \sigma_2 \to \sigma_1]\!] &= \mathsf{let\ z} : \sigma_2\ \mathsf{in}\ [\![\mathsf{t} : \sigma_1]\!] \\ [\![\mathsf{t}_1\ (\mathsf{t}_2 : \exists\,\bar{\beta}.\,\sigma_2) : \rho_1]\!] &= \exists\,\bar{\beta}.([\![\mathsf{t}_1 : \sigma_2 \to \rho_1]\!] \wedge [\![\mathsf{t}_2 : \sigma_2]\!]) \\ &\qquad\qquad \bar{\beta} \notin \mathsf{ftv}(\rho_1) \\ [\![\mathsf{let\ z} = (\mathsf{t}_1 : \exists\,\bar{\beta}.\,\sigma_1) & \\ \mathsf{in\ t}_2 : \rho_2]\!] &= \mathsf{let\ z} : \forall\bar{\beta}[[\![\mathsf{t}_1 : \sigma_1]\!]].\sigma_1\ \mathsf{in}\ [\![\mathsf{t}_2 : \rho_2]\!] \\ [\![\mathsf{t} : \forall\,\alpha.\,\sigma]\!] &= \forall\,\alpha.[\![\mathsf{t} : \sigma]\!] \end{aligned}$$

The solver of [PR05] need only be extended to solve subtyping constraints. These can easily be reduced to equality constraints by repeatedly applying the rules of Figure 7. (We have left it implicit in the first and second rules that variables α_1 and α_2 should not appear free on the left-hand side). Each rule preserves the meaning of constraints, indeed. When no rule applies, we are left with equality constraints of the form $\tau = \tau'$, which are treated as first-order unification constraints (the predicate $\leq$ may be interpreted as equality on monotypes). Hence, rules of Figure 7 are meant to be added to rules for solving first-order constraints as well as structural rules for rearranging constraints (see [PR05]). Indeed, the rules of Figure 7 can already be found in [OL96].

Lemma 9. *The rewriting rules of Figure 7 preserve constraint equivalence.*

Once the (exposed) subtyping constraints have been resolved, the remaining constraints are equality constraints between monotypes, which can then be resolved by a unification algorithm, as in [PR05]. The whole simplification process ends with a constraint in solved from, which determines a most general solution to the initial problem. Remark that a type substitution φ may always be represented as the type constraint $\wedge_{\alpha \in dom(\varphi)}(\alpha = \alpha\varphi)$.

Constraint generation rules

We now describe a type inference algorithm by turning type checking problems into type constraint problems, which can then be simplified as described above.

Constraint generation rules are given in Figure 8. They take as input a type expression t and an expected type σ and return a constraint that describes the conditions under which t may be assigned type σ. Maybe surprisingly,

Figure 9. Syntax-directed typing rules for $\mathsf{F}^{v}_{\mathsf{ML}}(\leq_{p}^{\eta-})$

$$\textsc{Var-Inst-Rho} \qquad \textsc{Gen}^{v} \qquad \textsc{Fun}$$

$\textsc{App}_{\mathsf{A}}{}^{v}$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \sigma_2[\bar\tau/\bar\beta] \to \forall\,\bar\alpha.\,\rho_1 \qquad \Gamma \vdash \mathsf{t}_2 : \sigma_2[\bar\tau/\bar\beta] \qquad \bar\alpha \notin \mathsf{ftv}^{\mathsf{t}_1\ \mathsf{t}_2}(\rho_1)}{\Gamma \vdash \mathsf{t}_1\ (\mathsf{t}_2 : \exists\,\bar\beta.\,\sigma_2) : \forall\,\bar\alpha.\,\rho_1}$$

$\textsc{Let}_{\mathsf{A}}\text{-}\textsc{Gen}^{v}$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \sigma_1[\bar\tau/\bar\beta] \qquad \Gamma, \mathsf{z} : \forall\,^{\mathsf{t}_1}\backslash\Gamma.\,\sigma_1[\bar\tau/\bar\beta] \vdash \mathsf{t}_2 : \forall\,\bar\alpha.\,\rho_2 \qquad \bar\alpha \notin \mathsf{ftv}^{\mathsf{t}_1,\mathsf{t}_2}(\rho_2)}{\Gamma \vdash \mathsf{let}\ \mathsf{z} = (\mathsf{t}_1 : \exists\,\bar\beta.\,\sigma_1)\ \mathsf{in}\ \mathsf{t}_2 : \forall\,\bar\alpha.\,\rho_2}$$

constraint generation does not take a typing context Γ: if the program t has free program identifiers x, so will the constraint $[\![\mathsf{t} : \sigma]\!]$. Then, the constraint may only be valid when placed in a context of the form $\mathsf{let}\ \Gamma\ \mathsf{in}\ \cdot$ that will assign types to free type variables of t.

Most constraint generation rules can be read straightforwardly and follow syntax-directed typing rules. An identifier x has type ρ is and only if ρ is an instance of the type of x. A function $\lambda\mathsf{z}\,.\,\mathsf{t}$ has type α if and only if it has type $\beta_2 \to \beta_1$ where α is of the form $\beta_2 \to \beta_1$ for some types β_1, β_2; a function $\lambda\mathsf{z}\,.\,\mathsf{t}$ has a type scheme $\sigma_2 \to \sigma_1$ if and only if t has type σ_1 assuming z has type scheme σ_2. The decomposition of applications is also straightforward. To decompose a let-binding, we first generate a constraint for typechecking its argument; this constraint is used to build a constrained type-scheme by generalizing all type variables $\bar\beta$ that are free in σ_1, but free in Γ. After simplification of $[\![\mathsf{t}_1 : \sigma_1]\!]$ some variables of $\bar\beta$ will in general be further constrained, maybe so that they can only be monotypes. Generalizing them is not a problem, since the resolution algorithm will take care of such dependences, which is one of the elegance of using type constraints. Finally, an expression t has a type $\forall\,\alpha.\,\sigma$ if and only if t has type σ for any instance of α.

A type checking problem $\Gamma \vdash \mathsf{t} : \sigma$ is equivalent to the constraint $\mathsf{let}\ \Gamma\ \mathsf{in}\ [\![\mathsf{t} : \sigma]\!]$ (which is syntactic sugar for a sequence of let constraints, as described in Figure 6). This is stated very precisely and concisely by the following two lemmas. This uses a correspondence mapping an idempotent substitution φ to a solved type constraint $\bigwedge_{\alpha \in dom(\varphi)} \alpha = \alpha\varphi$.

Lemma 10 (soundness). *If $\varphi \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ [\![\mathsf{t} : \sigma]\!]$, then $\Gamma\varphi \vdash \mathsf{t} : \sigma\varphi$.*

Lemma 11 (completeness). *If $\Gamma\varphi \vdash \mathsf{t} : \sigma\varphi$, then $\varphi \Vdash \mathsf{let}\ \Gamma\ \mathsf{in}\ [\![\mathsf{t} : \sigma]\!]$.*

We can read back these two theorems into a more traditional (but not really simpler) formulation.

Corollary 12. *Given a typing environment Γ, a program t, and a type scheme σ, a substitution φ is a solution to the type checking problem $\Gamma \vdash \mathsf{t} : \sigma$ if and only if it is a solution to the constraint $\mathsf{let}\ \Gamma\ \mathsf{in}\ [\![\mathsf{t} : \sigma]\!]$.*

2.5 Type inference with side-effects

As we have seen in Section 1.5, choosing a call-by-value semantics and extending the language with side-effects also implies some changes in the typing rules. Let $\mathsf{F}^{v}_{\mathsf{ML}}(\leq_{p}^{\eta-})$ be the language whose expressions and typing rules are the same as those of $\mathsf{F}_{\mathsf{ML}}(\leqslant)$, except for Rule Gen, which is

Figure 10. New constraint generation rules for $\mathsf{F}^{v}_{\mathsf{ML}}(\leq_{p}^{\eta-})$

$$\begin{array}{lr}
[\![\mathsf{t}_1\ (\mathsf{t}_2 : \exists\,\bar\beta.\,\sigma_2) : \forall\,\bar\alpha.\,\rho_1]\!] \ = & \\
\ \exists\,\bar\beta.([\![\mathsf{t}_1 : \sigma_2 \to \forall\,\bar\alpha.\,\rho_1]\!] \wedge [\![\mathsf{t}_2 : \sigma_2]\!]) & \bar\alpha = \emptyset \text{ if } \mathsf{t}_1\ \mathsf{t}_2 \in \mathcal{U} \\
[\![\mathsf{let}\ \mathsf{z} = (\mathsf{u}_1 : \exists\,\bar\beta.\,\sigma_1)\ \mathsf{in}\ \mathsf{t}_2 : \forall\,\bar\alpha.\,\rho_2]\!] \ = & \\
\ \mathsf{let}\ \mathsf{z} : \forall\bar\beta[[\![\mathsf{u}_1 : \sigma_1]\!]].\sigma_1\ \mathsf{in}\ [\![\mathsf{t}_2 : \forall\,\bar\alpha.\,\rho_2]\!] & \bar\alpha = \emptyset \text{ if } \mathsf{t}_2 \in \mathcal{U} \\
[\![\mathsf{let}\ \mathsf{z} = (\mathsf{t}_1 : \exists\,\bar\beta.\,\sigma_1)\ \mathsf{in}\ \mathsf{t}_2 : \forall\,\bar\alpha.\,\rho_2]\!] \ = & \\
\ \exists\,\bar\beta.([\![\mathsf{t}_1 : \sigma_1]\!] \wedge \mathsf{let}\ \mathsf{z} : \sigma_1\ \mathsf{in}\ [\![\mathsf{t}_2 : \forall\,\bar\alpha.\,\rho_2]\!]) & \mathsf{t}_1 \notin \mathcal{U} \\
[\![\mathsf{u} : \forall\,\alpha.\,\sigma]\!] \ = \ \forall\,\alpha.[\![\mathsf{u} : \sigma]\!] &
\end{array}$$

replaced by Rule $\textsc{Gen}^{v}$. We may extend the definition of non-expansive expressions given in Section 1.5 with $\mathsf{let}\ \mathsf{z} = \mathsf{u}_1\ \mathsf{in}\ \mathsf{u}_2$. The relation $\leq_{p}^{\eta-}$ is a subrelation of $\leq^{\eta-}$. Hence, $\mathsf{F}^{v}_{\mathsf{ML}}(\leq_{p}^{\eta-})$ is sound.

Preparing for type inference, we must also review the syntax-directed typing rules. Of course, we should change Gen to $\textsc{Gen}^{v}$. Since rule $\textsc{Gen}^{v}$ is more restrictive, only non-negative type variables may be generalized a posteriori in the type of an expansive expression. As a consequence, the result of the application (see Rule $\textsc{App}_{\mathsf{A}}$) may have a polymorphic type but only if this polymorphism comes from the codomain σ_1 of the type of t_1 or from non-negative free type variables in σ_1. Hence, we change Rule $\textsc{App}_{\mathsf{A}}$-Rho for Rule $\textsc{App}_{\mathsf{A}}{}^{v}$ to allow exactly those variables $\bar\alpha$ that do not belong to $\mathsf{ftv}^{\mathsf{t}_1\ \mathsf{t}_2}(\rho_1)$ in the type of the application—others may still be generalized afterward. The notation $\mathsf{ftv}^{\bar{\mathsf{t}}}(\sigma)$ where $\bar{\mathsf{t}}$ is a sequence of expressions stands for $\mathsf{ftv}(\sigma)$ if $\bar{\mathsf{t}} \in \mathcal{U}$ (*i.e.* all expressions of $\bar{\mathsf{t}}$ are non-expansive) and $\mathsf{ftv}^{\dagger}(\sigma)$ otherwise. Similarly, we change Rule $\textsc{Let}_{\mathsf{A}}$-Gen-Rho to $\textsc{Let}_{\mathsf{A}}\text{-}\textsc{Gen}^{v}$ which allows $\bar\alpha$ to be polymorphic in the type of t_2. The sequence of expressions $\mathsf{t}_1, \mathsf{t}_2$ used in the superscript of ftv controls the expansiveness of $\mathsf{let}\ \mathsf{z} = (\mathsf{t}_1 : \exists\,\bar\beta.\,\sigma_1)\ \mathsf{in}\ \mathsf{t}_2$. Consistently with the restriction of generalization, we have also replaced $\forall\,\backslash\Gamma.\,\sigma_1[\tau/\bar\beta]$ by $\forall\,^{\mathsf{t}_1}\backslash\Gamma.\,\sigma_1[\bar\tau/\bar\beta]$. The notation $\forall\,^{\mathsf{t}}\backslash\Gamma.\,\sigma$ stands for $\forall\,\mathsf{ftv}^{\mathsf{t}}(\sigma) \setminus \mathsf{ftv}(\Gamma).\,\sigma$. That is, generalizable variables in the type of t_1 are $\mathsf{ftv}(\sigma) \setminus \mathsf{ftv}(\Gamma)$ as before if t_1 is non-expansive, and $\mathsf{ftv}^{\dagger}(\sigma) \setminus \mathsf{ftv}(\Gamma)$ otherwise.

Constraint generation

Let us first consider the simpler case of the standard value restriction. That is, we ignore non-negative type variables which amounts to taking the empty set for $\mathsf{ftv}^{\dagger}(\sigma)$. In this case, $\mathsf{ftv}^{\bar{\mathsf{t}}}(\sigma)$ is $\mathsf{ftv}(\sigma)$ if all expressions of $\bar{\mathsf{t}}$ are non-expansive and is empty otherwise.

It is then straightforward to extend constraint generation: we need not add new forms of constraints but only test for expansiveness during constraint generation and generate different constraints for non-expansive applications and let-bindings. The modified rules are summarized in Figure 10 (capture-avoiding side-conditions have been omitted); these should replace the corresponding rules in Figure 8. Constraint resolution then proceeds as before. Lemmas 6, 7, 10, and 11 extend to $\mathsf{F}^{v}_{\mathsf{ML}}(\leq_{p}^{\eta-})$ with *standard* value restriction.

By lack of space, we refer to the full version [Rém05] for a discussion of the *enhanced* value restriction, which requires enriching the language of type-constraints.

2.6 Type inference for $\mathsf{F}_{\mathsf{ML}}(\leq_{p}^{\eta})$

Unsurprisingly, rules of Figure 5 do not form a syntax-directed presentation of $\mathsf{F}_{\mathsf{ML}}(\leq_{p}^{\eta})$: after replacing $\leq_{p}^{\eta-}$ by $\leq_{p}^{\eta}$ in Rule Var-Inst-Rho they would defined a typing relation that does not satisfy Rule Inst. Fortunately, we may easily reduce type inference for $\mathsf{F}_{\mathsf{ML}}(\leq_p^\eta)$ to type inference for $\mathsf{F}_{\mathsf{ML}}(\leq_p^{\eta-})$ by first putting typing problems into prenex-forms. More precisely, let $\mathsf{prf}(\exists\,\bar\beta.\,\sigma)$ be $\exists\,\bar\beta.\,\mathsf{prf}(\sigma)$ and $\mathsf{prf}(\mathsf{t})$ be a copy of t where all annotations θ have be replaced by their prenex-forms $\mathsf{prf}(\theta)$.

LEMMA 13. *Judgments* $\Gamma \vdash_{\mathsf{F}_{\mathsf{ML}}(\leq_p^\eta)} \mathsf{t} : \sigma$ *and* $\mathsf{prf}(\Gamma) \vdash_{\mathsf{F}_{\mathsf{ML}}(\leq_p^{\eta-})} \mathsf{prf}(\mathsf{t}) : \mathsf{prf}(\sigma)$ *are equivalent.*

This shows that the choice between $\leq_p^{\eta-}$ and $\leq_p^\eta$ is unimportant in predicative systems.

2.7 Recovering impredicativity

An important limitation of $\mathsf{F}_{\mathsf{ML}}(\leq_p^{\eta-})$ is its predicativity. Fortunately, there is an easy way to recover impredicativity by adding an explicit impredicative decidable fragment of type-containment, say $\leqslant_I$ to the implicit predicative type-containment relation $\leq_p^{\eta-}$ or $\leq_p^\eta$, say $\leqslant_P$. Indeed, to obtain full impredicativity, *i.e.*, in the end, the expressiveness of F, it suffices that $\leqslant_I$ be a larger relation than $\leq^{\mathsf{F}}$. We choose the relation $\leq_\approx^{\mathsf{F}}$, which is better suited for type inference, as it disregards the order of quantifiers.

Formally, we may simply introduce a collection of coercion functions (*i.e.* functions that evaluate to the identity) $(\_ : \exists\,\bar\beta.\,\sigma_1 \triangleright \sigma_2)$ with types $\forall\,\bar\beta.\,\sigma_1 \to \sigma_2$ for all pairs (σ_1,σ_2) in $\leqslant_I$ and with $\bar\beta$ equal to $\mathsf{ftv}(\sigma_1) \cup \mathsf{ftv}(\sigma_2)$. As for annotations, we require coercions to be closed, although this is only a matter of simplification. This is the purpose of the existentially bound variables whose scope extends to both sides of the coercion. We refer to this language as $\mathsf{F}_{\mathsf{ML}}(\leqslant_I, \leqslant_P)$.

We may here take advantage of constrained type schemes and be slightly more flexible. Let us introduce new forms of constraints $\sigma_1 \leqslant \sigma_2$ whose meaning is given by $\phi \Vdash \sigma_1 \leqslant \sigma_2$ if and only if $\sigma_1\phi \leqslant \sigma_2\phi$ (for some given relations $\leqslant$). Then, we may assign $(\_ : \exists\,\bar\beta.\,\sigma_1 \triangleright \sigma_2)$ the constrained type scheme $\forall\bar\beta[\sigma_1 \leqslant_I \sigma_2].\sigma_1 \to \sigma_2$. This has two advantages: first, it internalizes the verification of the $\leqslant_I$ type-containment; second, it allows to provide types σ_1 and σ_2 up to some monomorphic instantiation. For example, taking $\leq^{\mathsf{F}}$ for $\leqslant_I$, the coercion $(\_ : \exists\,\beta.\,\forall\,\alpha.\,\alpha \to \beta \to \alpha \triangleright \sigma \to \tau \to \sigma)$ is now valid for any (closed) type τ and type scheme σ; finding out that β must actually be τ can be left to type inference.

This extension is safe by construction, as long as $\leqslant_I$ is a subrelation of $\leq^\eta$, since it amounts to giving the identity a valid type in $\mathsf{F}(\leq^\eta)$. Regarding type inference, it only remains to solve $\leqslant_I$ type-containment constraints so that the coercion is well-defined and Rule VAR-INST-RHO applies.

Let us now focus on the particular case of $\mathsf{F}_{\mathsf{ML}}(\leq_\approx^{\mathsf{F}}, \leq_p^{\eta-})$, which we shall abbreviate as F_{ML}. Type constraints $\sigma_1 \leq_\approx^{\mathsf{F}} \sigma_2$ can be reduced to $\mathsf{canon}(\sigma_1) \leq^{\mathsf{F}} \mathsf{canon}(\sigma_2)$. So we are left to solving constraints of the form $\sigma_1 \leq^{\mathsf{F}} \sigma_2$. Recall that all instances of the relation $\leq^{\mathsf{F}}$ are of the form $\forall\,\bar\alpha.\,\rho \leq^{\mathsf{F}} \forall\,\bar\gamma.\,\rho[\bar\sigma/\bar\alpha]$ with $\bar\gamma \notin \mathsf{ftv}(\forall\,\bar\alpha.\,\rho)$. Thus, the constraint $\forall\,\bar\alpha.\,\rho_1 \leq^{\mathsf{F}} \forall\,\bar\gamma.\,\rho_2$ with free type variables $\bar\beta$ is equivalent to the existence of type schemes $\bar\sigma$ such that ρ_2 and $\rho_1[\bar\sigma/\bar\alpha]$ are equal modulo α-conversion with variables $\bar\gamma$ not in $\mathsf{ftv}(\forall\,\bar\alpha.\,\rho_1)$. That is, it is equivalent to the equality $\forall\,\bar\gamma.\,\exists\,\bar\alpha.(\rho_1 = \rho_2)$ where $\bar\alpha$ range over type schemes and $\bar\alpha$ and $\bar\beta$ range over monotypes. This is a (restricted) form of unification modulo α-conversion and under a mixed prefix (but no β-reduction is ever involved). Solving such constraints is folklore knowledge. We refer the reader to the full version for details.

Coercion functions are meant to be applied to terms. Formally, an application requires a type annotation, which in this case may be exactly the domain of the coercion. We may avoid repeating the annotation by letting the syntactic sugar $(\mathsf{t} : \exists\,\bar\beta.\,\sigma_1 \triangleright \sigma_2)$ stand for $(\_ : \exists\,\bar\beta.\,\sigma_1 \triangleright \sigma_2)\ (\mathsf{t} : \exists\,\bar\beta.\,\sigma_1)$. When a coercion is in application position, it still needs an annotation, even though the coercion may already carries all the necessary type information. Hence, we may see $\mathsf{t}_1\ (\mathsf{t}_2 : \exists\,\bar\beta.\,\sigma_1 \triangleright \sigma_2)$ as syntactic sugar for $\mathsf{t}_1\ ((\mathsf{t}_2 : \exists\,\bar\beta.\,\sigma \triangleright \sigma') : \exists\,\bar\beta.\,\sigma')$.

There still seem to be some redundancies in coercions since the polymorphic shape of the expected type σ_2' must be provided when typechecking a coercion $(\mathsf{t} : \exists\,\bar\beta.\,\sigma_1 \triangleright \sigma_2)$. However, σ_2' is only known up to $\leq_p^{\eta-}$. That is, in general, we only have $\sigma_2 \leq \sigma_2'$. If we were to take σ_2' for σ_2, we would then be letf with $\leq_\approx^{\mathsf{F}} \circ \leq_p^{\eta-}$ constraints. However, we have not explored yet the resolution of such constraints.

Expressiveness The set of raw terms typable in $\mathsf{F}_{\mathsf{ML}}(\leqslant_I, \leqslant_P)$ is exactly $\mathsf{F}(\leqslant_I \circ \leqslant_P)$. Hence, F_{ML} contains terms of both F and $\mathsf{F}(\leq_p^{\eta-})$—but not all terms of $\mathsf{F}(\leq^\eta)$. One may in fact generalize the idea of explicit coercions by providing typing evidence for any coercion, which can be done by writing coercions in F_{ML}. We could thus also allow coercions of the form $(\_ : \mathsf{t})$ where t is a term of F_{ML} that η-reduces to the identity. This way, we would finally reach all of $\mathsf{F}(\leq^\eta)$. However, writing coercion functions explicitly is rather heavy.

An alternative way to recover impredicativity is to use semi-explicit polymorphism [GR97], which simplifies significantly here, as types of the host language are already of arbitrary rank.

3. Propagation of annotations in $\mathsf{F}_{\mathsf{ML}}^?$

We consider the language F_{ML}. However, we first treat coercions as constants, as if we were in $\mathsf{F}_{\mathsf{ML}}(\leq_p^{\eta-})$. We shall discuss special elaboration of coercions in Section 3.5 and elaboration in $\mathsf{F}_{\mathsf{ML}}^v(\leq_p^{\eta-})$ in Section 3.4.

Many type annotations may seem redundant in F_{ML}. For instance, consider the expression $\mathsf{let\ f} = (\lambda \mathsf{z}\,.\,\mathsf{z}\ \mathsf{z} : \sigma_{id} \to \sigma_{id})\ \mathsf{in\ f}\ (\lambda \mathsf{y}\,.\,\mathsf{y} : \sigma_{id})$, which is well typed, but becomes ill-typed if we remove the annotation on the application. However, one may argue that given the type of f, it is *obvious* what the annotation on the application should be.

We first show that only the polymorphic skeletons of annotations, where the monomorphic leaves are stripped off, actually matter. We refer to them as *shapes*. Computation on shapes is simpler than computation on types, because shapes are closed. We exploit this to propose a preprocessing step that propagates shapes before typechecking in F_{ML}.

3.1 Shapes

Let *shapes* be closed polymorphic types extended with a unary type constructor $\sharp$. Shapes are considered modulo the *absorbing equation* $\sharp \to \sharp = \sharp$. A shape is in canonical form when it contains the minimum number of occurrences of $\sharp$. The shape of a polymorphic type σ, written $\lceil\sigma\rceil$ is obtained from σ by replacing all free type variables of σ by $\sharp$. We use $\mathcal{S}$ to range over shapes. Shapes capture the polymorphic structure of types. For example, consider the type σ_0 equal to $\forall\,\alpha_1.(\forall\,\alpha_2.(\alpha_1 \to \alpha_2) \to (\beta_0 \to \beta_0)) \to (\beta_1 \to \beta_2)$. Its shape $\lceil\sigma\rceil$ is $\forall\,\alpha_1.(\forall\,\alpha_2.(\alpha_1 \to \alpha_2) \to (\sharp \to \sharp)) \to (\sharp \to \sharp)$, which can be put in its canonical form $\forall\,\alpha_1.\,(\forall\,\alpha_2.\,(\alpha_1 \to \alpha_2) \to \sharp) \to \sharp$. By extension, the shape of an annotation $\lceil\exists\,\bar\beta.\,\sigma\rceil$ is simply $\lceil\sigma\rceil$. A shape $\mathcal{S}$ may be read back as a type annotation, written $\lfloor\mathcal{S}\rfloor$. By definition $\lfloor\mathcal{S}\rfloor$ is $\exists\,\bar\alpha.\,\sigma$ if $\mathcal{S}$ is in canonical form and syntactically equal to $\sigma[\sharp/\bar\alpha]$, and each variable of $\bar\alpha$ occurs exactly once in σ. For example, the annotation $\lfloor\lceil\sigma\rceil\rfloor$ is $\exists\beta_1,\beta_2\,\forall\,\alpha_1.\,(\forall\,\alpha_2.\,(\alpha_1 \rightarrow \alpha_2) \rightarrow \beta_1) \;\rightarrow\; \beta_2$. We sometimes need to strip off the front quantifiers of a shape $\mathcal{S}$. However, the resulting type may contain free type variables and must be reshaped. We write $\mathcal{S}^\flat$ for $\lceil\rho\rceil$ where $\lfloor\mathcal{S}\rfloor$ is of the form $\exists\,\bar\beta.\forall\,\bar\alpha.\,\rho$.

Our interest for shapes is that only shapes of annotations matter for type inference. If we write $\lfloor\lceil\mathsf{t}\rceil\rfloor$ for the term t where each annotation θ has been replaced by $\lfloor\lceil\theta\rceil\rfloor$, this property is precisely captured by the following lemma.

Lemma 14. *If* $\Gamma \vdash \mathsf{t} : \sigma$ *then* $\Gamma \vdash \lfloor\lceil\mathsf{t}\rceil\rfloor : \sigma$.

Since shapes are a projection of types, one could project typing rules as well, and obtain a “shaping” relation that would assign shapes to programs. By construction, one would then expect a property such as if $\Gamma \vdash \mathsf{t} : \sigma$ then $\lceil\Gamma\rceil \vdash \mathsf{t} : \lceil\sigma\rceil$. However, such a result would not be very useful, since we already have an algorithm for typechecking.

Conversely, one could hope for some kind of shape-inference algorithm and by combination with typechecking at a given shape to solve type inference problems at unknown shapes. However, shape-inference is of course not quite realistic if we define it as finding *every* shape for which a type could be inferred. However, we may give up completeness, or even soundness, and look for some *obvious* shape for which a type might be inferred. Incompleteness may not be a problem in the context of type reconstruction where we attempt to rebuild missing type annotations in an *obvious* manner and accept to fail if there is no way to do so. *Since type reconstruction is not going to be complete with respect to some simple logical specification, let it be at least* simple *and* intuitive*!*

Thus, we shall now allow unannotated application nodes $\mathsf{t}_1\ \mathsf{t}_2$ and let-binding nodes $\mathsf{let}\ \mathsf{z} = \mathsf{t}_1\ \mathsf{in}\ \mathsf{t}_2$—here, we mean no annotation at all and not the trivial annotation $\exists\,\beta.\,\beta$, which we may still write, but *explicitly*. Since the expected shape may now be missing, it becomes convenient to simultaneously allow explicit annotations on formal parameters of functions, which we write $\lambda\mathsf{z} : \theta\,.\,\mathsf{t}$. Precisely, let expressions of the language $\mathsf{F}^?_{\mathsf{ML}}$ be those of F_{ML} extended with the three constructions above. Finally, we may now define typechecking in $\mathsf{F}^?_{\mathsf{ML}}(\leqslant)$ by elaboration into terms of F_{ML} computing on shapes to fill in the missing annotations. This is a form of (incomplete) shape inference that returns both a shape and an elaborated term of F_{ML}. We write shape inference using judgments of the form $\Gamma \vdash_\uparrow \mathsf{t} : \mathcal{S} \Rightarrow \mathsf{t}'$ where the $\uparrow$ is there to remember that $\mathcal{S}$ is inferred. Because $\mathsf{F}^?_{\mathsf{ML}}$ is a superset of F_{ML}, there are still cases where the expected type, hence the expected shape, are known. In such cases, elaboration may be turned into a checking mode: it then uses the given shape and elaborates subterms from which it can return an elaborated term. Hence, we recursively define a judgment $\Gamma \vdash_\downarrow \mathsf{t} : \mathcal{S} \Rightarrow \mathsf{t}'$. Here, Γ, t and $\mathcal{S}$ are all given and t' is returned. The “$\downarrow$” sign indicates that $\mathcal{S}$ is only checked. The idea of mixing checking and inference modes is taken from Peyton Jones and Shields [PVWS05a] but goes back to older works such as local type inference [PT00]. The direction of arrows is taken from [PVWS05a].

3.2 Elaboration

The two elaboration judgments are defined by the set of rules of Figure 11. We used variable ϵ to range over $\uparrow$ and $\downarrow$. This allows factoring some rules that could otherwise be written as pairs of rules.

Figure 11. Elaboration rules for $\mathsf{F}^?_{\mathsf{ML}}$

Var-C
$$\frac{\mathsf{x} : \mathcal{S}' \in \Gamma}{\Gamma \vdash_\downarrow \mathsf{x} : \mathcal{S} \Rightarrow \mathsf{x}}$$

Var-I
$$\frac{\mathsf{x} : \mathcal{S} \in \Gamma}{\Gamma \vdash_\uparrow \mathsf{x} : \mathcal{S} \Rightarrow \mathsf{x}}$$

Fun-C
$$\frac{\Gamma, \mathsf{z} : \mathcal{S}_2 \vdash_\downarrow \mathsf{t} : \mathcal{S}_1 \Rightarrow \mathsf{t}'}{\Gamma \vdash_\downarrow \lambda\mathsf{z}\,.\,\mathsf{t} : \mathcal{S}_2 \rightarrow \mathcal{S}_1 \Rightarrow \lambda\mathsf{z}\,.\,\mathsf{t}'}$$

Fun-I
$$\frac{\Gamma, \mathsf{z} : \sharp \vdash_\uparrow \mathsf{t} : \mathcal{S} \Rightarrow \mathsf{t}'}{\Gamma \vdash_\uparrow \lambda\mathsf{z}\,.\,\mathsf{t} : \sharp \rightarrow \mathcal{S} \Rightarrow \lambda\mathsf{z}\,.\,\mathsf{t}'}$$

Fun$_{\mathsf{A}}$-C
$$\frac{\Gamma, \mathsf{z} : \lceil\sigma\rceil \vdash_\downarrow \mathsf{t} : \mathcal{S}_1 \Rightarrow \mathsf{t}'}{\begin{array}{l}\Gamma \vdash_\downarrow \lambda\mathsf{z} : \exists\,\bar\beta.\,\sigma\,.\,\mathsf{t} : \mathcal{S}_2 \rightarrow \mathcal{S}_1 \\ \Rightarrow \lambda\mathsf{z}\,.\,\mathsf{let}\ \mathsf{z} = (\mathsf{z} : \exists\,\bar\beta.\,\sigma)\ \mathsf{in}\ \mathsf{t}'\end{array}}$$

Fun$_{\mathsf{A}}$-I
$$\frac{\Gamma, \mathsf{z} : \lceil\sigma\rceil \vdash_\uparrow \mathsf{t} : \mathcal{S} \Rightarrow \mathsf{t}'}{\begin{array}{l}\Gamma \vdash_\uparrow \lambda\mathsf{z} : \exists\,\bar\beta.\,\sigma\,.\,\mathsf{t} : \lceil\sigma\rceil \rightarrow \mathcal{S} \\ \Rightarrow \lambda\mathsf{z}\,.\,\mathsf{let}\ \mathsf{z} = (\mathsf{z} : \exists\,\bar\beta.\,\sigma)\ \mathsf{in}\ \mathsf{t}'\end{array}}$$

Let-ϵ
$$\frac{\Gamma \vdash_\uparrow \mathsf{t}_1 : \mathcal{S}_1 \Rightarrow \mathsf{t}_1' \qquad \Gamma, \mathsf{z} : \mathcal{S}_1 \vdash_\epsilon \mathsf{t}_2 : \mathcal{S}_2 \Rightarrow \mathsf{t}_2'}{\Gamma \vdash_\epsilon \mathsf{let}\ \mathsf{z} = \mathsf{t}_1\ \mathsf{in}\ \mathsf{t}_2 : \mathcal{S}_2 \Rightarrow \mathsf{let}\ \mathsf{z} = (\mathsf{t}_1' : \lfloor\mathcal{S}_1\rfloor)\ \mathsf{in}\ \mathsf{t}_2'}$$

Let$_{\mathsf{A}}$-ϵ
$$\frac{\Gamma \vdash_\downarrow \mathsf{t}_1 : \lceil\sigma\rceil \Rightarrow \mathsf{t}_1' \qquad \Gamma, \mathsf{z} : \lceil\sigma\rceil \vdash_\epsilon \mathsf{t}_2 : \mathcal{S}_2 \Rightarrow \mathsf{t}_2'}{\Gamma \vdash_\epsilon \mathsf{let}\ \mathsf{z} = (\mathsf{t}_1 : \exists\,\bar\beta.\,\sigma)\ \mathsf{in}\ \mathsf{t}_2 : \mathcal{S}_2 \Rightarrow \mathsf{let}\ \mathsf{z} = (\mathsf{t}_1' : \exists\,\bar\beta.\,\sigma)\ \mathsf{in}\ \mathsf{t}_2'}$$

App$_{\mathsf{A}}$-C
$$\frac{\Gamma \vdash_\downarrow \mathsf{t}_1 : \lceil\sigma\rceil \rightarrow \mathcal{S} \Rightarrow \mathsf{t}_1' \qquad \Gamma \vdash_\downarrow \mathsf{t}_2 : \lceil\sigma\rceil \Rightarrow \mathsf{t}_2'}{\Gamma \vdash_\downarrow \mathsf{t}_1\ (\mathsf{t}_2 : \exists\,\bar\beta.\,\sigma) : \mathcal{S} \;\Rightarrow \mathsf{t}_1'\ (\mathsf{t}_2' : \exists\,\bar\beta.\,\sigma)}$$

App$_{\mathsf{A}}$-I
$$\frac{\Gamma \vdash_\uparrow \mathsf{t}_1 : \mathcal{S} \Rightarrow \mathsf{t}_1' \qquad \mathcal{S}^\flat = \mathcal{S}_2 \rightarrow \mathcal{S}_1 \qquad \Gamma \vdash_\downarrow \mathsf{t}_2 : \lceil\sigma\rceil \Rightarrow \mathsf{t}_2'}{\Gamma \vdash_\uparrow \mathsf{t}_1\ (\mathsf{t}_2 : \exists\,\bar\beta.\,\sigma) : \mathcal{S}_1 \;\Rightarrow \mathsf{t}_1'\ (\mathsf{t}_2' : \exists\,\bar\beta.\,\sigma)}$$

App-C
$$\frac{\Gamma \vdash_\uparrow \mathsf{t}_1 : \mathcal{S} \Rightarrow \mathsf{t}_1' \qquad \mathcal{S}^\flat = \mathcal{S}_2 \rightarrow \mathcal{S}_1' \qquad \Gamma \vdash_\uparrow \mathsf{t}_2 : \mathcal{S}_2' \Rightarrow \mathsf{t}_2'}{\Gamma \vdash_\downarrow \mathsf{t}_1\ \mathsf{t}_2 : \mathcal{S}_1 \;\Rightarrow \mathsf{t}_1'\ (\mathsf{t}_2' : \lfloor\mathcal{S}_2\rfloor)}$$

App-I
$$\frac{\Gamma \vdash_\uparrow \mathsf{t}_1 : \mathcal{S} \Rightarrow \mathsf{t}_1' \qquad \mathcal{S}^\flat = \mathcal{S}_2 \rightarrow \mathcal{S}_1 \qquad \Gamma \vdash_\uparrow \mathsf{t}_2 : \mathcal{S}_2' \Rightarrow \mathsf{t}_2'}{\Gamma \vdash_\uparrow \mathsf{t}_1\ \mathsf{t}_2 : \mathcal{S}_1 \;\Rightarrow \mathsf{t}_1'\ (\mathsf{t}_2' : \lfloor\mathcal{S}_2\rfloor)}$$

Rule Var-C checks that there is a binding for x and ignores the shape of x. Of course, a certain relation should hold between $\mathcal{S}'$ and $\mathcal{S}$. However, this will be verified by later typechecking in F_{ML}, so we may just ignore this condition. Rule Var$_{\mathsf{A}}$-I reads the shape from the environment Γ. Note that the best known shape of x is $\mathcal{S}$ and not an instance of $\mathcal{S}$, which would be weaker.

In Rule Fun-C the given shape of the conclusion provides us with both the shape of the parameter z and the expected shape of t. We can thus elaborate the premise in checking mode. Rule Fun-I is just the opposite, since nothing is known from the conclusion. We thus use shape $\sharp$ for the parameter and elaborate the premise in inference mode. In Rule Fun$_{\mathsf{A}}$-C we actually have extra information since the shape of the parameter is known from both the annotation in the conclusion and the shape of the conclusion. Again, some relation between $\lceil\sigma\rceil$ and $\mathcal{S}_2$ should hold. We use $\lceil\sigma\rceil$, which is explicitly given and simply ignore $\mathcal{S}_2$ during elaboration. Still, the let-binding in the elaborated term, which stands for an additional pure type-annotation, ensures that the correct relation between σ and $\mathcal{S}_2$ will be verified. Rule Fun$_{\mathsf{A}}$-I, is similar except that the premise is called in inference instead of checking mode.

Cases for let-bindings (Let-ϵ and Let$_{\mathsf{A}}$-ϵ) are all very similar. The left-premise is called in inference mode when there is no annotation on t_1; the right-premise is called in inference mode when the conclusion is itself in inference mode. The important detail is that in all cases, variable z is bound to the best-known-shape $\lceil\sigma\rceil$ or $\mathcal{S}_1$, whether it is taken from the annotation or inferred. In particular, one cannot "generalize the shape" in any meaningful way. Indeed, the type inferred for t_1 during typechecking may have free variables that could later be generalized during typechecking. However, we cannot tell during elaboration, so we may only assume for the shape of z the best known shape of t_1. Of course, typechecking may later do better!

An annotated application is elaborated in checking mode (Rule App$_\mathsf{A}$-C) by calling both premises in checking mode, since all necessary shapes are known. In inference mode (App$_\mathsf{A}$-I), the left-premise must be called in inference mode because the expected shape of t_1 is not entirely known. However, only the range of the inferred shape $\mathcal{S}_1$ matters and is used in the elaboration (remember that $\mathcal{S}^\flat$ is $\mathcal{S}$ stripped off its toplevel quantifiers and reshaped). Again the domain $\mathcal{S}_2$ should be related to σ in some way, but we may ignore it here. Rules App-I and App-C deal with the inference mode. Both premises must be called in inference mode, since none of the expected shapes is ever entirely known. The annotation $\mathcal{S}_2$ is however taken from the left-hand side. This choice is somehow arbitrary, but it seems to usually work better in practice. The difference between inference and checking modes is that the return shape $\mathcal{S}_1$ is given in checking mode, while it is the range of $\mathcal{S}^\flat$ in inference mode. (In checking mode the range $\mathcal{S}_1'$ of $\mathcal{S}^\flat$, which is ignored, should be related to $\mathcal{S}_1$ in some way.)

By construction, elaboration always keep existing annotations, so it is the identity on terms of $\mathsf{F_{ML}}$ and idempotent for terms of $\mathsf{F}^?_{\mathsf{ML}}$.

Variations In typing rules for unannotated applications, we have somehow made arbitrary choices, taking the annotation from the domain of the inferred shape of the function and discarding the inferred shape of the argument. In [Rém05], we propose other elaboration rules for applications that may also sometimes take the shape of the argument for building the annotation.

3.3 Typechecking in $\mathsf{F}^?_{\mathsf{ML}}$

Of course, elaboration may return ill-typed programs, since shapes are only an approximation of types. Hence the programs resulting from elaboration must be submitted to type inference in $\mathsf{F_{ML}}$. Actually, well-typedness in $\mathsf{F}^?_{\mathsf{ML}}$ is simply defined by means of elaboration in $\mathsf{F_{ML}}$.

Definition 1. Let $\Gamma \vdash_\epsilon \mathsf{t} : \sigma$ if and only if there exists an expression t' such that $\lceil\Gamma\rceil \vdash_\epsilon \mathsf{t} : \lceil\sigma\rceil \Rightarrow \mathsf{t}'$ and $\Gamma \vdash \mathsf{t}' : \sigma$. □

By construction, $\mathsf{F}^?_{\mathsf{ML}}$ is sound. Elaboration preserves the type erasure, hence the semantics of terms.

Elaboration rules are given in syntax-directed form and are deterministic. They straightforwardly define an algorithm that, given Γ and t as input, returns the shape of t and an elaborated term t'. Thus, type inference problems in $\mathsf{F}^?_{\mathsf{ML}}$ can be solved as follows: given Γ and t, let elaboration compute both a term t' and a shape $\mathcal{S}$ such that $\Gamma \vdash_\uparrow \mathsf{t} : \mathcal{S} \Rightarrow \mathsf{t}'$; let $\exists\bar\beta.\sigma$ be $\lfloor\mathcal{S}\rfloor$; infer a principal substitution φ for the type checking problem $\Gamma \vdash \mathsf{t}' : \sigma$ in $\mathsf{F_{ML}}$ and returns the substitution φ and the type $\forall\setminus\Gamma\varphi.\,\sigma\varphi$. By construction this type inference algorithm is sound and complete with respect to Definition 1, because propagation is deterministic and type-checking in $\mathsf{F_{ML}}$ is sound and complete for the logical specification of $\mathsf{F_{ML}}$.

One may argue that Definition 1 does not have a logical flavor. Indeed, we agree! However, we sustain the claim that it is simple and easy to understand. Ill-typed programs may be explained by providing both the elaborated program *and* an explanation of why it does not typecheck in $\mathsf{F_{ML}}$. Either the elaborated program is not as expected, and the user should easily see why since elaboration is simple or he should understand the type error with respect to the elaborated program.

3.4 Elaboration with coercions

So far, we have elaborated coercions as constants. However, we may take advantage of elaboration to allow partial type information on explicit coercions as well. Let us allow to omit any side of an explicit coercion, *i.e.* to write $(\mathsf{t} : \exists\bar\beta.\ \triangleright\sigma_2)$, $(\mathsf{t} : \exists\bar\beta.\,\sigma_1\triangleright)$ and $(\mathsf{t} : \triangleright)$ respectively. Elaboration must return a fully specified coercion by filling in the missing part. We do this in the obvious ways, turning a coercion into a simple type annotation or simply discarding it when insufficient information is available. By lack of space, we refer to the full version for the corresponding elaboration rules.

Conversely, we may also allow elaboration to insert coercions that are not user-specified. However, in general, this would amount to guessing polymorphism. So we should only insert obvious coercions. In particular, when otherwise elaboration would lead to a typechecking error. There are such opportunities in rules Var-C, Fun$_\mathsf{A}$-C, and several application rules. We say that the annotation $\mathcal{S}_1$ is compatible with the annotation $\mathcal{S}_2$, which we write $\mathcal{S}_1 \leq_p^{\eta-} \mathcal{S}_2$, if there exists type schemes σ_1 and σ_2 of shape $\mathcal{S}_1$ and $\mathcal{S}_2$ such that $\sigma_1 \leq_p^{\eta-} \sigma_2$. For instance, $(\forall\alpha.\,\alpha) \to \sharp$ is not compatible with $(\forall\alpha.\,\alpha\to\alpha) \to \sharp$.

We may restrict rule Var-C to cases where the shape $\mathcal{S}'$ of x is $\sharp$ or is compatible with the expected shape $\mathcal{S}$. Otherwise, we may elaborate x as if it were $(\mathsf{x} : \triangleright)$. Similarly, we may restrict Fun$_\mathsf{A}$-C to cases where $\mathcal{S}_2'$ is $\sharp$ or compatible with $\lceil\sigma\rceil$. Otherwise, we may elaborate $\lambda\mathsf{z} : \exists\bar\beta.\,\sigma\,.\,\mathsf{t}$ as if it were $\lambda\mathsf{z}\,.\,\mathsf{let}\ \mathsf{z} = (\mathsf{z} : \exists\bar\beta.\ \triangleright\sigma)\ \mathsf{in}\ \mathsf{t}$. Rules for applications have several places where elaboration may obviously lead to a typechecking failure. We may introduce a coercion on the function-side or on the argument-side. Or on both sides, but this would require guessing an intermediate type scheme at which both coercions would meet. Further investigation is necessary to find a reasonable and useful strategy for inserting coerions around applications.

3.5 Elaboration with an imperative semantics

Actually, elaboration does not depend on the differences between $\mathsf{F_{ML}}(\leq_p^{\eta-})$ and $\mathsf{F}^v_{\mathsf{ML}}(\leq_p^{\eta-})$. To safely elaborate programs for $\mathsf{F}^v_{\mathsf{ML}}(\leq_p^{\eta-})$, we only need to replace typechecking in $\mathsf{F_{ML}}(\leq_p^{\eta-})$ by typechecking in $\mathsf{F}^v_{\mathsf{ML}}(\leq_p^{\eta-})$. We write $\mathsf{F}^{v\,?}_{\mathsf{ML}}$ for the corresponding language. This language is safe by construction, since elaboration preserves the semantics and the final typechecking ensure type-safety. Of course, type-safety is not sufficient to make $\mathsf{F}^v_{\mathsf{ML}}(\leq_p^{\eta-})$ an interesting language. One potential problem is that elaborated programs would then too often be rejected because of the value-restriction.

For instance, consider an expression t_1 of the form $(\lambda\mathsf{z}\,.\,\lambda\mathsf{y}\,.\,\mathsf{y})\ \mathsf{t}_0$ where t_0 is a well-typed expansive expression. Let σ_{id} be $\forall\alpha.\,\alpha\to\alpha$. In $\mathsf{F}^?_{\mathsf{ML}}$ we have both $\vdash_\uparrow \mathsf{t}_1 : \sigma_{id}$ and $\vdash_\downarrow \mathsf{t}_1 : \sigma_{id}$. In $\mathsf{F}^{v\,?}_{\mathsf{ML}}$, we do not have $\vdash_\uparrow \mathsf{t}_1 : \sigma_{id}$ any more, because this will first infer a monomorphic type and fail to generalize at the end. Fortunately, we still have $\vdash_\downarrow \mathsf{t}_1 : \sigma_{id}$. Note that if t_0 were non-expansive, then t_1 would also be non-expansive and no annotation would be needed—just to emphasize the benefits of using a larger class of non-expansive expressions than just values and variables.

As another example consider the expression t_2 equal to $\mathsf{let}\ \mathsf{f} = (\lambda \mathsf{y} . \mathsf{y} : \sigma_{id})\ \mathsf{in}\ (\lambda \mathsf{z} . \mathsf{f})\ \mathsf{t}_0$. Here, we even have $\vdash_\uparrow \mathsf{t}_2 : \sigma_{id}$ because the polymorphic shape of f can be inferred. One may wonder why an explicit annotation on f is needed, since $\mathsf{F}^?_{\mathsf{ML}}$ can infer that $\lambda \mathsf{y} . \mathsf{y}$ has polymorphic type σ_{id}. Indeed, the problem is that elaboration is performed before type inference. A solution is to use incremental elaboration as described in Section 3.6

3.6 Incremental elaboration

Annotating let-bound expressions with ML types has shown to be sometimes useful. This may seem surprising, since these types can actually be inferred. The reason is that elaboration, which uses annotations, is performed before typechecking and does not see inferred let-bound polymorphism. Incremental elaboration is a solution to this problem. Instead of performing elaboration of the whole program followed by typechecking we may elaborate and typecheckeck programs by smaller parts, such as toplevel phrases. Consider, for instance, the expression $\mathsf{let}\ \mathsf{z}_1 = \mathsf{t}_1\ \mathsf{in}\ \ldots \mathsf{let}\ \mathsf{z}_n = \mathsf{t}_n\ \mathsf{in}\ \mathsf{t}$. It can be seen as a succession of $n+1$ phrases, each of which but the last one augmenting the initial environment with a binding $\mathsf{z}_k : \sigma_k$ where σ_k is the type inferred for t_k.

Assuming such a mechanism, no ML-like annotation would ever be useful on the toplevel bindings of $\bar{\mathsf{z}}$. We may push this idea further and apply it to local bindings as well using the tree-structure of let-bindings to order the sequence of small elaboration followed by typechecking steps. However, this is getting (slightly) more complicated and loosing the simplicity of the two-step mechanism so that at the end the user may not so easily understand the elaboration process. Incremental elaboration at the level of toplevel phrases seems to be a good compromise.

4. Related works and conclusions

We have already widely discussed related works in the introduction and in particular the most closely related one, PVWS, that inspired this work. Another close work, developed in parallel with ours, is a recent proposal by Peyton-Jones *et al.* [VWP05], which we refer to below as VWP. It contributes a significant improvement over PVWS.

Leaving impredicativity aside, $\mathsf{F}_{\mathsf{ML}}(\leq_p^{\eta-})$ and PVWS have similar, but incomparable, expressiveness. That is, both have the capability to propagate annotations in both directions. However, they also differ in small details and there are examples that one can type and the other cannot. Since our elaboration is defined as a simple preprocessing step, it can easily be modified, *e.g.* to match PVWS more closely, but not entirely.

We have chosen to elaborate applications without propagation of information from the function to the argument (*e.g.* App-I), as opposed to PVWS. Our choice is more symmetric and keeps the flow of elaboration bottom-up, which is more intuitive for the user to guess the elaborated program. Of course, we lose some information this way, at the benefice of more predictable elaboration and typechecking. Actually, the result of elaboration is then passed to typechecking, so some information may still flow sideway, *e.g.* from the function side to the argument side. However, such information does not depend on underneath first-order unification nor on the order in which typechecking is performed. This is actually another limit of our separation of elaboration and typechecking. Incremental elaboration re-introduces some ordering in which typechecking must be performed, but in a controlled and intuitive manner.

The language VWP shares a lot with PVWS including the monolithic algorithmic specification of typechecking. However, VWP also improves over PVWS in at least two significant ways. First, checking and inference modes are carried on types rather than on typing judgments, much as for colored local type inference [OZZ01]. This provides with a more precise control of these modes. For instance, it allows to specify that some part of a type is known and to be checked while some other part is still unknown and to be inferred—a capability that we missed in the elaboration of applications. However, as a result, VWP is also quite involved and it seems quite difficult to guess in advance whether a program is typable without running the algorithm, which can hardly be done mentally or even on paper.

Second, VWP also allows for impredicative polymorphism. The treatment of impredicative polymorphism seems better integrated in VWP. However, it is also deeply hidden into algorithmic inference rules and an ad-hoc multi-argument typechecking rule for applications. This makes it difficult to understand the interaction between impredicative instantiation and predicative type-containment, since both seem to be left implicit, which clearly cannot be the case.

The monolithic type inference process of VWP seems to be more powerful in propagating known information than $\mathsf{F}^?_{\mathsf{ML}}$. However, it simultaneously mixes complicated orthogonal concepts in hard-wired algorithmic rules, and as PVWS, it lacks a simple specification.

We see at least three directions for future improvements. Improving the expressiveness of the core language is to be favored. Finding a stronger relation than $\leq_p^\eta$ for which first-order constraints would be easily solvable. In particular solving constraints of the form $\leq_p^{\eta-} \circ \leq^{\mathsf{F}} \circ \leq_p^{\eta-}$ would enable explicit impredicative instantiation up to predicative containment by taking this relation for $\leqslant_I$. We should also explore properties of elaborations in more details, which we may then advantageously exploit. Finally, elaboration could probably be made more accurate by introducing variables ranging over type schemes to represent unknown information and avoid assigning them $\sharp$ a priori. This could also help with the elaboration of impredicative coercions. Hopefully, such solutions may avoid the need for more incremental elaboration, which could be confusing and less intuitive for the user. However, we might then lose our original goal to keep type inference within the resolution of first-order constraints.

If we are to lose this simplicity, we should also compare with ML^{F} [LBR03, LB04], which was designed so as to allow impredicative instantiation from the beginning, and does it rather well, by comparison with impredicative instantiation in $\mathsf{F}^?_{\mathsf{ML}}$. Conversely, ML^{F} does not have type-containment—and does not allow $\leq_p^\eta$ instantiations—but this does not seem to be a problem at all in practice. Already present in ML^{F} is the idea of elaboration to propagate annotations from interfaces to abstractions inside expressions, although elaboration used for ML^{F} remains a rather trivial process. ML^{F} has obviously a more powerful type inference engine and seems to perform better than $\mathsf{F}^?_{\mathsf{ML}}$. However its metatheory is also more involved.

Conclusions

In summary, we have proposed a core language F_{ML} with first-order type inference with predicative type containment and explicit impredicative type-instantiation. A more convenient surface language $\mathsf{F}^{?}_{\mathsf{ML}}$ may be used to alleviate some repetitions of type annotations in source programs. It is defined by a simple elaboration procedure into the core language.

We believe that the decomposition of type inference into a simple elaboration procedure that propagates second-order type annotations followed by a first-order ML-like type inference mechanism is a good compromise between a logical and an algorithmic presentation. At least, it clearly separates the algorithmic elaboration process, which is complete by definition, but incomplete by nature, from the logical order-independent specification of type inference.

As observed, small variations in the logical specification, *e.g.* in the type-containment relation or the application of generalization may result in rather different type inference algorithms. This confirms, if it were necessary, that algorithmic specifications of type inference are fragile. Of course, algorithmic elaboration is a remedy. The goal remains to find expressive type systems with simple logical specifications for which we have complete inference algorithms.

Acknowledgments

I would like to particularly thank François Pottier for many fruitful discussions regarding this work. I am also grateful to Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones for their useful feedback concerning this paper and, more generally, for discussions on type inference with type-containment.

References

[BCP99] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999.

[Gar04] Jacques Garrigue. Relaxing the value restriction. In *International Symposium on Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, Nara, April 2004. Springer-Verlag.

[GR97] Jacques Garrigue and Didier Rémy. Extending ML with semi-explicit higher-order polymorphism. In Takayasu Ito and Martín Abadi, editors, *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 20–46. Springer-Verlag, September 1997.

[HP99] Haruo Hosoya and Benjamin C. Pierce. How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, June 1999.

[LB04] Didier Le Botlan. *MLF: Une extension de ML avec polymorphisme de second ordre et instanciation implicite*. PhD thesis, University of Paris 7, June 2004. (english version).

[LBR03] Didier Le Botlan and Didier Rémy. MLF: Raising ML to the power of system-F. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 27–38, August 2003.

[Lei91] Daniel Leivant. Finitely stratified polymorphism. *Information and Computation*, 93(1):93–113, July 1991.

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

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

[OL96] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *ACM Symposium on Principles of Programming*, pages 54–67, St. Petersburg, Florida, January 21–24, 1996. ACM Press.

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

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

[PR] François Pottier and Didier Rémy. The essence of ML type inference. Extended version of [PR05] (in preparation).

[PR05] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[PS03] Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. Submitted to ICFP 2004, March 2003.

[PT00] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[PVWS05a] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. Submitted to the Journal of Functional Programming, June 2005.

[PVWS05b] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. technical appendix. Private communication, June 2005.

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

[Rém05] Didier Rémy. Simple, partial type-inference for System F based on type-containment. Full version, September 2005.

[TU02] Jerzy Tiuryn and Pawel Urzyczyn. The subtyping problem for second-order types is undecidable. *Information and Computation*, 179(1):1–18, 2002.

[VWP05] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy type inference for higher-rank types and impredicativity. Available electronically at , April 2005.

[Wel94] Joe B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 176–185, 1994.

[Wel95] Joe B. Wells. The undecidability of Mitchell's subtyping relation. Technical Report 95-019, Computer Science Department, Boston Univiversity, December 1995.

[Wel96] Joe B. Wells. Typability is undecidable for F+eta. Technical Report 96-022, Computer Science Department, Boston Univiversity, March 1996.

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