A Church-Style Intermediate Language for ML^F

Didier Rémy[1] Boris Yakobowski[2]

[1] INRIA Paris - Rocquencourt
[2] CEA, LIST, Laboratoire Sureté des Logiciels,
Boîte 94, 91191 Gif-sur-Yvette Cedex, France.

Abstract ML^F is a type system that seamlessly merges ML-style implicit but second-class polymorphism with System-F explicit first-class polymorphism. We present $x\mathsf{ML}^\mathsf{F}$, a Church-style version of ML^F with full type information that can easily be maintained during reduction. All parameters of functions are explicitly typed and both type abstraction and type instantiation are explicit. However, type instantiation in $x\mathsf{ML}^\mathsf{F}$ is more general than type application in System F. We equip $x\mathsf{ML}^\mathsf{F}$ with a small-step reduction semantics that allows reduction in any context and show that this relation is confluent and type preserving. We also show that both subject reduction and progress hold for weak-reduction strategies, including call-by-value with the value-restriction.

Introduction

ML^F (Le Botlan and Rémy 2003, 2007; Rémy and Yakobowski 2008b) is a type system that seamlessly merges ML-style implicit but second-class polymorphism with System-F explicit first-class polymorphism. This is done by enriching System-F types. Indeed, System F is not well-suited for partial type inference, as illustrated by the following example. Assume that a function, say choice, of type $\forall\,(\alpha)\ \alpha \to \alpha \to \alpha$ and the identity function id, of type $\forall\,(\beta)\ \beta \to \beta$, have been defined. How can the application choice to id be typed in System F? Should choice be applied to the type $\forall\,(\beta)\ \beta \to \beta$ of the identity that is itself kept polymorphic? Or should it be applied to the monomorphic type $\gamma \to \gamma$, with the identity being applied to γ (where γ is bound in a type abstraction in front of the application)? Unfortunately, these alternatives have incompatible types, respectively $(\forall\,(\alpha)\ \alpha \to \alpha) \to (\forall\,(\alpha)\ \alpha \to \alpha)$ and $\forall\,(\gamma)\ (\gamma \to \gamma) \to (\gamma \to \gamma)$: none is an instance of the other. Hence, in System F, one is forced to irreversibly choose between one of the two explicitly typed terms.

However, a type inference system cannot choose between the two, as this would sacrifice completeness and be somehow arbitrary. This is why ML^F enriches types with instance-bounded polymorphism, which allows to write more expressive types that factor out in a single type all typechecking alternatives in such cases as the example of choice. Now, the type $\forall\,(\alpha \geqslant \tau)\ \alpha \to \alpha$, which should be read "$\alpha \to \alpha$ *where* α *is any instance of* τ", can be assigned to choice id, and the two previous alternatives can be recovered *a posteriori* by choosing different instances for α.

Currently, the language ML^F comes with a Curry-style version $i\mathsf{ML}^\mathsf{F}$ where no type information is needed and a type-inference version $e\mathsf{ML}^\mathsf{F}$ that requires partial type information (Le Botlan and Rémy 2007). However, $e\mathsf{ML}^\mathsf{F}$ is not quite in Church's style, since a large amount of type information is still implicit and partial type information cannot be easily maintained during reduction. Hence, while $e\mathsf{ML}^\mathsf{F}$ is a good surface language, it is not a good candidate for use as an internal language during the compilation process, where some program transformations, and perhaps some reduction steps, are being performed. This has been a problem for the adoption of ML^F in the Haskell community (Peyton Jones 2003), as the Haskell compilation chain uses an explicitly-typed internal language.

This is also an obstacle to proving subject reduction, which does not hold in $e\mathsf{ML}^\mathsf{F}$. In a way, this is unavoidable in a language with non-trivial partial type inference. Indeed, type annotations cannot be completely dropped, but must at least be transformed and reorganized during reduction. Still, one could expect that $e\mathsf{ML}^\mathsf{F}$ be equipped with reduction rules for type annotations. This has actually been considered in the original presentation of ML^F, but only with limited success. The reduction kept track of annotation sites during reduction; this showed, in particular, that no new annotation site needs to be introduced during reduction. Unfortunately, the exact form of annotations could not be maintained during reduction, by lack of an appropriate language to describe their computation. As a result, it has only been shown that some type derivation can be rebuilt after the reduction of a well-typed program, but without exhibiting an algorithm to compute them during reduction.

Independently, Rémy and Yakobowski (2008b) have introduced graphic constraints, both to simplify the presentation of ML^F and to improve its type inference algorithm. This also lead to a simpler, more expressive definition of ML^F.

In this paper, we present $x\mathsf{ML}^\mathsf{F}$, a Church-style version of ML^F that contains full type information. In fact, type checking becomes a simple and local verification process—by contrast with type inference in $e\mathsf{ML}^\mathsf{F}$, which is based on unification. In $x\mathsf{ML}^\mathsf{F}$, type abstraction, type instantiation, and all parameters of functions are explicit, as in System F. However, type instantiation is more general and more atomic than type application in System F: we use explicit type instantiation expressions that are proof evidences for the type instance relations.

In addition to the usual β-reduction, we give a series of reduction rules for simplifying type instantiations. These rules are confluent when allowed in any context. Moreover, reduction preserves typings, and is sufficient to reduce all typable expressions to a value when used in either a call-by-value or call-by-name setting. This establishes the soundness of ML^F for a call-by-name semantics for the first time. Notably, $x\mathsf{ML}^\mathsf{F}$ is a conservative extension of System F.

The paper is organized as follows. We present $x\mathsf{ML}^\mathsf{F}$, its syntax and its static and dynamic semantics in §1. We study its main properties, including type soundness for different evaluations strategies in §2. We discuss possible variations, as well as related and future works in §3. All proofs are omitted, but can be found in (Yakobowski 2008, Chapters 14 & 15).

$\alpha, \beta, \gamma, \delta$		**Type variable**	x, y, z		**Term variable**
τ	::=	**Type**	a	::=	**Term**
	α	Type variable		x	Variable
	$\tau \to \tau$	Arrow type		$\lambda(x : \tau)\ a$	Function
	$\forall\,(\alpha \geqslant \tau)\ \tau$	Quantification		$a\ a$	Application
	$\bot$	Bottom type		$\Lambda(\alpha \geqslant \tau)\ a$	Type function
ϕ	::=	**Instantiation**		$a\,\phi$	Instantiation
	τ	Bottom		$\mathsf{let}\ x = a\ \mathsf{in}\ a$	Let-binding
	$!\alpha$	Abstract	Γ	::=	**Environment**
	$\forall\,(\geqslant \phi)$	Inside		$\varnothing$	Empty
	$\forall\,(\alpha \geqslant)\ \phi$	Under		$\Gamma, \alpha \geqslant \tau$	Type variable
	$\&$	$\forall$-elimination		$\Gamma, x : \tau$	Term variable
	$⅋$	$\forall$-introduction			
	$\phi; \phi$	Composition			
	$\mathbb{1}$	Identity			

Figure 1. Grammar of types, instantiations, and terms

1 The calculus

Types, instantiations, terms, and typing environments All the syntactic definitions of $x\mathsf{ML}^\mathsf{F}$ can be found in Figure 1. We assume given a countable collection of variables ranged over by letters α, β, γ, and δ. As usual, types include type variables and arrow types. Other type constructors will be added later—straightforwardly, as the arrow constructor receives no special treatment. Types also include a bottom type $\bot$ that corresponds to the System-F type $\forall\alpha.\alpha$. Finally, a type may also be a form of bounded quantification $\forall\,(\alpha \geqslant \tau)\ \tau'$, called *flexible* quantification, that generalizes the $\forall\alpha.\tau$ form of System F and, intuitively, restricts the variable α to range only over instances of τ. The variable α is bound in τ' but not in τ. (We may write $\forall\,(\alpha)\ \tau'$ when the bound τ is $\bot$.)

In Church-style System F, type instantiation inside terms is simply type application, of the form $a\,\tau$. By contrast, type instantiation $a\,\phi$ in $x\mathsf{ML}^\mathsf{F}$ details every intermediate instantiation step, so that it can be checked locally. Intuitively, the *instantiation* ϕ transforms a type τ into another type τ' that is an instance of τ. In a way, ϕ is a witness for the instance relation that holds between τ and τ'. It is therefore easier to understand instantiations altogether with their static semantics, which will be explained in the next section.

Terms of $x\mathsf{ML}^\mathsf{F}$ are those of the λ-calculus enriched with let constructs, with two small differences. Type instantiation $a\,\phi$ generalizes System-F type application. Type abstractions are extended with an instance bound τ and written $\Lambda(\alpha \geqslant \tau)\ a$. The type variable α is bound in a, but not in τ. We abbreviate $\Lambda(\alpha \geqslant \bot)\ a$ as $\Lambda(\alpha)\ a$, which simulates the type abstraction $\Lambda\alpha.\,a$ of System F. We write $\mathsf{ftv}(\tau)$ and $\mathsf{ftv}(a)$ the set of type variables that appear free in τ and a.

We identify types, instantiations, and terms up to the renaming of bound variables. The capture-avoiding substitution of a variable v inside an expression s by an expression s' is written $s\{v \leftarrow s'\}$.

$$\textsc{Inst-Bot}\ \frac{}{\Gamma \vdash \tau : \bot \leq \tau} \qquad \textsc{Inst-Under}\ \frac{\Gamma, \alpha \geqslant \tau \vdash \phi : \tau_1 \leq \tau_2}{\Gamma \vdash \forall\,(\alpha \geqslant)\ \phi : \forall\,(\alpha \geqslant \tau)\ \tau_1 \leq \forall\,(\alpha \geqslant \tau)\ \tau_2} \qquad \textsc{Inst-Abstr}\ \frac{\alpha \geqslant \tau \in \Gamma}{\Gamma \vdash !\alpha : \tau \leq \alpha}$$

$$\textsc{Inst-Inside}\ \frac{\Gamma \vdash \phi : \tau_1 \leq \tau_2}{\Gamma \vdash \forall\,(\geqslant \phi) : \forall\,(\alpha \geqslant \tau_1)\ \tau \leq \forall\,(\alpha \geqslant \tau_2)\ \tau} \qquad \textsc{Inst-Intro}\ \frac{\alpha \notin \mathsf{ftv}(\tau)}{\Gamma \vdash \invamp : \tau \leq \forall\,(\alpha \geqslant \bot)\ \tau}$$

$$\textsc{Inst-Comp}\ \frac{\Gamma \vdash \phi_1 : \tau_1 \leq \tau_2 \quad \Gamma \vdash \phi_2 : \tau_2 \leq \tau_3}{\Gamma \vdash \phi_1; \phi_2 : \tau_1 \leq \tau_3} \qquad \textsc{Inst-Elim}\ \frac{}{\Gamma \vdash \& : \forall\,(\alpha \geqslant \tau)\ \tau' \leq \tau'\{\alpha \leftarrow \tau\}} \qquad \textsc{Inst-Id}\ \frac{}{\Gamma \vdash \mathbb{1} : \tau \leq \tau}$$

Figure 2. Type instance

As usual, type environments assign types to program variables. However, instead of just listing type variables, as is the case in System F, type variables are also assigned a bound in a binding of the form $\alpha \geqslant \tau$. We write $\mathsf{dom}(\Gamma)$ for the set of all terms and type variables that are bound by Γ. We also assume that typing environments are *well-formed*, *i.e.* they do not bind twice the same variable and free type variables appearing in a type of the environment Γ must be bound earlier in Γ. Formally, the empty environment is well-formed and, given a (well-formed) environment Γ, the relations $\alpha \notin \mathsf{dom}(\Gamma)$ and $\mathsf{ftv}(\tau) \subseteq \mathsf{dom}(\Gamma)$ must hold to form environments $\Gamma, \alpha \geqslant \tau$ and $\Gamma, x : \tau$.

Instantiations Instantiations ϕ are defined in Figure 1. Their typing, described in Figure 2, are *type instance* judgments of the form $\Gamma \vdash \phi : \tau \leq \tau'$, stating that in environment Γ, the instantiation ϕ transforms the type τ into the type τ'.

The *bottom* instantiation τ expresses that (any) type τ is an instance of the bottom type. The *abstract* instantiation $!\alpha$, which assumes that the hypothesis $\alpha \geqslant \tau$ is in the environment, abstracts the bound τ of α as the type variable α. The *inside* instantiation $\forall\,(\geqslant \phi)$ applies ϕ to the bound τ' of a flexible quantification $\forall\,(\alpha' \geqslant \tau')\ \tau$. Conversely, the *under* instantiation $\forall\,(\alpha \geqslant)\ \phi$ applies ϕ to the type τ under the quantification. The type variable α is bound in ϕ; the environment in the premise of the rule Inst-Under is increased accordingly. The *quantifier introduction* $\invamp$[3] introduces a fresh trivial quantification $\forall\,(\alpha \geqslant \bot)$. Conversely, the *quantifier elimination* $\&$ eliminates the bound of a type of the form $\forall\,(\alpha \geqslant \tau)\ \tau'$ by substituting τ for α in τ'. This amounts to definitely choosing the present bound τ for α, while the bound before the application could be further instantiated by some inside instantiation. The *composition* $\phi; \phi'$ witnesses the transitivity of type instance, while the *identity* instantiation $\mathbb{1}$ witnesses reflexivity.

[3] The choice of $\invamp$ is only by symmetry with the elimination form $\&$ described next, and has no connection at all with linear logic.

$$
\begin{array}{llll}
\tau\ (!\alpha) = \alpha & & \tau\ \wp = \forall\,(\alpha \geqslant \bot)\ \tau & \alpha \notin \mathsf{ftv}(\tau) \\
\bot\ \tau = \tau & (\forall\,(\alpha \geqslant \tau)\ \tau')\ \& = \tau'\{\alpha \leftarrow \tau\} & & \\
\tau\ \mathbb{1} = \tau & (\forall\,(\alpha \geqslant \tau)\ \tau')\ (\forall\,(\geqslant \phi)) = \forall\,(\alpha \geqslant \tau\,\phi)\ \tau' & & \\
\tau\ (\phi_1; \phi_2) = (\tau\,\phi_1)\,\phi_2 & (\forall\,(\alpha \geqslant \tau)\ \tau')\ (\forall\,(\alpha \geqslant)\ \phi) = \forall\,(\alpha \geqslant \tau)\ (\tau'\,\phi) & &
\end{array}
$$

Figure 3. Type instantiation (on types)

Example Let τ_{min}, τ_{cmp}, and τ_{and} be the types of the parametric minimum and comparison functions and of the conjunction of boolean formulas:

$$
\tau_{\mathsf{min}} \triangleq \forall\,(\alpha \geqslant \bot)\ \alpha \to \alpha \to \alpha \qquad \tau_{\mathsf{cmp}} \triangleq \forall\,(\alpha \geqslant \bot)\ \alpha \to \alpha \to \mathsf{bool}
$$
$$
\tau_{\mathsf{and}} \triangleq \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}
$$

Let ϕ be the instantiation $\forall\,(\geqslant \mathsf{bool}); \&$. Then, $\vdash\ \phi : \tau_{\mathsf{min}} \leq \tau_{\mathsf{and}}$ and $\vdash\ \phi : \tau_{\mathsf{cmp}} \leq \tau_{\mathsf{and}}$ hold. Let τ_K be the type $\forall\,(\alpha \geqslant \bot)\ \forall\,(\beta \geqslant \bot)\ \alpha \to \beta \to \alpha$ (*e.g.* of the λ-term $\lambda(x)\ \lambda(y)\ x$) and ϕ' be the instantiation[4] $\forall\,(\alpha \geqslant)\ (\forall\,(\geqslant \alpha); \&)$. Then, $\phi' : \tau_K \leq \tau_{\mathsf{min}}$.

Type application As above, we often instantiate a quantification over $\bot$ and immediately substitute the result. Moreover, this pattern corresponds to the System-F unique instantiation form. Therefore, we define $\langle\tau\rangle$ as syntactic sugar for $(\forall\,(\geqslant \tau); \&)$. The instantiations ϕ and ϕ' can then be abbreviated as $\langle\mathsf{bool}\rangle$ and $\forall\,(\alpha \geqslant)\ \langle\alpha\rangle$. More generally, we write $\langle\phi\rangle$ for the computation $(\forall\,(\geqslant \phi); \&)$.

Properties of instantiations Since instantiations make all steps in the instance relation explicit, their typing is deterministic.

Lemma 1. *If* $\Gamma \vdash \phi\colon \tau \leq \tau_1$ *and* $\Gamma' \vdash \phi\colon \tau \leq \tau_2$, *then* $\tau_1 = \tau_2$.

The use of Γ' instead of Γ may be surprising. However, Γ does not contribute to the instance relation, except in the side condition of rule Inst-Abstr. Hence, the type instance relation defines a partial function, called *type instantiation*[5] that, given an instantiation ϕ and a type τ, returns (if it exists) the unique type $\tau\,\phi$ such that $\vdash \phi : \tau \leq \tau\,\phi$. An inductive definition of this function is given in Figure 3. Type instantiation is complete for type instance:

Lemma 2. *If* $\Gamma \vdash \phi : \tau \leq \tau'$, *then* $\tau\,\phi = \tau'$.

However, the fact that $\tau\,\phi$ may be defined and equal to τ' does not imply that $\Gamma \vdash \phi : \tau \leq \tau'$ holds for some Γ. Indeed, type instantiation does not check the premise of rule Inst-Abstr. This is intentional, as it avoids parametrizing type instantiation over the type environment. This means that type instantiation is not sound *in general*. This is never a problem, however, since we only use type instantiation originating from well-typed terms for which there always exists some context Γ such that $\Gamma \vdash \phi : \tau \leq \tau'$.

[4] The occurrence of α in the inside instantiation is bound by the under instantiation.

[5] There should never be any ambiguity with the operation $a\,\phi$ on expressions; moreover, both operations have strong similarities.

$$\textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{Let}\ \frac{\Gamma \vdash a : \tau \qquad \Gamma, x : \tau \vdash a' : \tau'}{\Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \tau'} \qquad \textsc{App}\ \frac{\Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1}$$

$$\textsc{Abs}\ \frac{\Gamma, x : \tau \vdash a : \tau'}{\Gamma \vdash \lambda(x : \tau)\ a : \tau \to \tau'} \qquad \textsc{TAbs}\ \frac{\Gamma, \alpha \geqslant \tau' \vdash a : \tau \qquad \alpha \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \Lambda(\alpha \geqslant \tau')\ a : \forall\,(\alpha \geqslant \tau')\ \tau} \qquad \textsc{TApp}\ \frac{\Gamma \vdash a : \tau \qquad \Gamma \vdash \phi : \tau \leq \tau'}{\Gamma \vdash a\,\phi : \tau'}$$

Figure 4. Typing rules for $x\mathsf{ML}^\mathsf{F}$

We say that types τ and τ' are equivalent in Γ if there exist ϕ and ϕ' such that $\Gamma \vdash \phi : \tau \leq \tau'$ and $\Gamma \vdash \phi' : \tau' \leq \tau$. Although types of $x\mathsf{ML}^\mathsf{F}$ are *syntactically* the same as the types of $i\mathsf{ML}^\mathsf{F}$—the Curry-style version of ML^F (Le Botlan and Rémy 2007)—they are richer, because type equivalence in $x\mathsf{ML}^\mathsf{F}$ is finer than type equivalence in $i\mathsf{ML}^\mathsf{F}$, as will be explained in §3.

Typing rules for $x\mathsf{ML}^\mathsf{F}$ Typing rules are defined in Figure 4. Compared with System F, the novelties are type abstraction and type instantiation, unsurprisingly. The typing of a type abstraction $\Lambda(\alpha \geqslant \tau)\ a$ extends the typing environment with the type variable α bound by τ. The typing of a type instantiation $a\,\phi$ resembles the typing of a coercion, as it just requires the instantiation ϕ to transform the type of a into the type of the result. Of course, it has the full power of the type application rule of System F. For example, the type instantiation $a\ \langle\tau\rangle$ has type $\tau'\{\alpha \leftarrow \tau\}$ provided the term a has type $\forall\,(\alpha)\ \tau'$. As in System F, a well-typed closed term has a unique type—in fact, a unique typing derivation.

A let-binding $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$ cannot entirely be treated as an abstraction for an immediate application $(\lambda(x : \tau_1)\ a_2)\ a_1$ because the former does not require a type annotation on x whereas the latter does. This is nothing new, and the same as in System F extended with let-bindings. (Notice however that τ_1, which is the type of a_1, is fully determined by a_1 and could be synthesized by a typechecker.)

Example Let id stand for the identity $\Lambda(\alpha \geqslant \bot)\ \lambda(x : \alpha)\ x$ and τ_id for the type $\forall\,(\alpha \geqslant \bot)\ \alpha \to \alpha$. We have $\vdash \mathsf{id} : \tau_\mathsf{id}$. The function choice mentioned in the introduction, may be defined as $\Lambda(\beta \geqslant \bot)\ \lambda(x : \beta)\ \lambda(y : \beta)\ x$. It has type $\forall\,(\beta \geqslant \bot)\ \beta \to \beta \to \beta$. The application of choice to id, which we refer to below as $\mathsf{choice\_id}$, may be defined as $\Lambda(\beta \geqslant \tau_\mathsf{id})\ \mathsf{choice}\ \langle\beta\rangle\ (\mathsf{id}\,(!\beta))$ and has type $\forall\,(\beta \geqslant \tau_\mathsf{id})\ \beta \to \beta$. The term $\mathsf{choice\_id}$ may also be given weaker types by type instantiation. For example, $\mathsf{choice\_id}\,\&$ has type $(\forall\,(\alpha \geqslant \bot)\ \alpha \to \alpha) \to (\forall\,(\alpha \geqslant \bot)\ \alpha \to \alpha)$ as in System F, while $\mathsf{choice\_id}\,(\otimes; \forall\,(\gamma \geqslant)\ (\forall\,(\geqslant \langle\gamma\rangle); \&))$ has the ML type $\forall\,(\gamma \geqslant \bot)\ (\gamma \to \gamma) \to \gamma \to \gamma$.

Reduction The semantics of the calculus is given by a small-step reduction semantics. We let reduction occur in any context, including under abstractions.

$$
\begin{array}{rlll}
(\lambda(x:\tau)\ a_1)\ a_2 & \longrightarrow a_1\{x \leftarrow a_2\} & & (\beta) \\
\mathsf{let}\ x = a_2\ \mathsf{in}\ a_1 & \longrightarrow a_1\{x \leftarrow a_2\} & & (\beta_{\mathsf{let}}) \\
a\,\mathbb{1} & \longrightarrow a & & (\iota\text{-Id}) \\
a\,(\phi;\phi') & \longrightarrow a\,\phi\,(\phi') & & (\iota\text{-Seq}) \\
a\,\invamp & \longrightarrow \Lambda(\alpha \geqslant \bot)\ a & \alpha \notin \mathsf{ftv}(a) & (\iota\text{-Intro}) \\
(\Lambda(\alpha \geqslant \tau)\ a)\,\& & \longrightarrow a\{!\alpha \leftarrow \mathbb{1}\}\{\alpha \leftarrow \tau\} & & (\iota\text{-Elim}) \\
(\Lambda(\alpha \geqslant \tau)\ a)\,(\forall\,(\alpha \geqslant)\ \phi) & \longrightarrow \Lambda(\alpha \geqslant \tau)\ (a\,\phi) & & (\iota\text{-Under}) \\
(\Lambda(\alpha \geqslant \tau)\ a)\,(\forall\,(\geqslant \phi)) & \longrightarrow \Lambda(\alpha \geqslant \tau\,\phi)\ a\{!\alpha \leftarrow \phi; !\alpha\} & & (\iota\text{-Inside}) \\
E[a] \longrightarrow E[a'] & \text{if}\ \ a \longrightarrow a' & & (\text{Context})
\end{array}
$$

Figure 5. Reduction rules

That is, the evaluation contexts are single-hole contexts, given by the grammar:

$$
\begin{array}{rcl}
E & ::= & [\,\cdot\,] \mid E\,\phi \mid \lambda(x:\tau)\ E \mid \Lambda(\alpha \geqslant \tau)\ E \\
& \mid & E\ a \mid a\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ a \mid \mathsf{let}\ x = a\ \mathsf{in}\ E
\end{array}
$$

The reduction rules are described in Figure 5. As usual, basic reduction steps contain β-reduction, with the two variants (β) and (β_{let}). Other basic reduction rules, related to the reduction of type instantiations and called ι-steps, are described below. The one-step reduction is closed under the context rule. We write $\longrightarrow_\beta$ and $\longrightarrow_\iota$ for the two subrelations of $\longrightarrow$ that contains only Context and β-steps or ι-step, respectively. Finally, the reduction is the reflexive and transitive closure $\longrightarrow\!\!\!\!\rightarrow$ of the one-step reduction relation.

Reduction of type instantiation Type instantiation redexes are all of the form $a\,\phi$. The first three rules do not constrain the form of a. The identity type instantiation is just dropped (Rule ι-Id). A type instantiation composition is replaced by the successive corresponding type instantiations (Rule ι-Seq). Rule ι-Intro introduces a new type abstraction in front of a; we assume that the bound variable α is fresh in a. The other three rules require the type instantiation to be applied to a type abstraction $\Lambda(\alpha \geqslant \tau)\ a$. Rule ι-Under propagates the type instantiation under the bound, inside the body a. By contrast, Rule ι-Inside propagates the type instantiation ϕ inside the bound, replacing τ by $\tau\,\phi$. However, as the bound of α has changed, the domain of the type instantiations $!\alpha$ is no more τ, but $\tau\,\phi$. Hence, in order to maintain well-typedness, all the occurrences of the instantiation $!\alpha$ in a must be simultaneously replaced by the instantiation $(\phi; !\alpha)$. Here, the instantiation $!\alpha$ is seen as atomic, *i.e.* all occurrences of $!\alpha$ are substituted, but other occurrences of α are left unchanged (see the appendix for the formal definition). For instance, if a is the term

$$\Lambda(\alpha \geqslant \tau)\ \lambda(x:\alpha \to \alpha)\ \lambda(y:\bot)\ y\,(\alpha \to \alpha)\ (z\,(!\alpha))$$

then, the type instantiation $a\,(\forall\,(\geqslant \phi))$ reduces to:

$$\Lambda(\alpha \geqslant \tau\,\phi)\ \lambda(x:\alpha \to \alpha)\ \lambda(y:\bot)\ y\,(\alpha \to \alpha)\ (z\,(\phi; !\alpha))$$

Rule ι-Elim eliminates the type abstraction, replacing all the occurrences of α inside a by the bound τ. All the occurrences of $!\alpha$ inside τ (used to instantiate τ

into α) become vacuous and must be replaced by the identity instantiation. For example, reusing the term a above, $a\,\&$ reduces to $\lambda(x : \tau \to \tau)\ \lambda(y : \bot)\ y\,(\tau \to \tau)\ (z\ \mathbb{1})$. Notice that type instantiations $a\,\tau$ and $a\,(!\alpha)$ are irreducible.

Examples of reduction Let us reuse the term $\mathsf{choice\_id}$ defined in §1 as $\Lambda(\beta \geqslant \tau_{\mathsf{id}})\ \mathsf{choice}\ \langle\beta\rangle\ (\mathsf{id}\,(!\beta))$. Remember that $\langle\tau\rangle$ stands for the System-F type application τ and expands to $(\forall\,(\geqslant\tau); \&)$. Therefore, the type instantiation $\mathsf{choice}\ \langle\beta\rangle$ reduces to the term $\lambda(x : \beta)\ \lambda(y : \beta)\ x$ by ι-Seq, ι-Inside and ι-Elim. Hence, the term $\mathsf{choice\_id}$ reduces by these rules, Context, and (β) to the expression $\Lambda(\beta \geqslant \tau_{\mathsf{id}})\ \lambda(y : \beta)\ \mathsf{id}\,(!\beta)$.

Below are three specialized versions of $\mathsf{choice\_id}$ (remember that $\forall\,(\alpha)\ \tau$ and $\Lambda(\alpha)\ a$ are abbreviations for $\forall\,(\alpha \geqslant \bot)\ \tau$ and $\Lambda(\alpha \geqslant \bot)\ a$). Here, all type instantiations are eliminated by reduction, but this is not always possible in general.

$$\begin{aligned}
\mathsf{choice_id}\ \langle\langle\mathsf{int}\rangle\rangle \quad &: \quad (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int}) \\
&\longrightarrow\!\!\!\!\!\rightarrow \lambda(y : \mathsf{int} \to \mathsf{int})\ (\lambda(x : \mathsf{int})\ x) \\
\mathsf{choice_id}\ \& \quad &: \quad (\forall\,(\alpha)\ \alpha \to \alpha) \to (\forall\,(\alpha)\ \alpha \to \alpha) \\
&\longrightarrow\!\!\!\!\!\rightarrow \lambda(y : \forall\,(\alpha)\ \alpha \to \alpha)\ (\Lambda(\alpha)\ \lambda(x : \alpha)\ x) \\
\mathsf{choice_id}\ (\otimes; \forall\,(\gamma \geqslant)\ (\forall\,(\geqslant \langle\gamma\rangle); \&)) \quad &: \quad \forall\,(\gamma)\ (\gamma \to \gamma) \to (\gamma \to \gamma) \\
&\longrightarrow\!\!\!\!\!\rightarrow \Lambda(\gamma)\ \lambda(y : \gamma \to \gamma)\ (\lambda(x : \gamma)\ x)
\end{aligned}$$

System F as a subsystem of $x\mathsf{ML}^{\mathsf{F}}$ System F can be seen as a subset of $x\mathsf{ML}^{\mathsf{F}}$, using the following syntactic restrictions: all quantifications are of the form $\forall\,(\alpha)\ \tau$ and $\bot$ is not a valid type anymore (however, as in System F, $\forall\,(\alpha)\ \alpha$ is); all type abstractions are of the form $\Lambda(\alpha)\ a$; and all type instantiations are of the form $a\ \langle\tau\rangle$.

The derived typing rule for $\Lambda(\alpha)\ a$ and $a\ \langle\tau\rangle$ are exactly the System-F typing rules for type abstraction and type application. Hence, typechecking in this restriction of $x\mathsf{ML}^{\mathsf{F}}$ corresponds to typechecking in System F.

Moreover, the reduction in this restriction also corresponds to reduction in System F. Indeed, a reducible type application is necessarily of the form $(\Lambda(\alpha)\ a)\,\langle\tau\rangle$ and can always be reduced to $a\{\alpha \leftarrow \tau\}$ as follows:

$$\begin{aligned}
(\Lambda(\alpha)\ a)\ \langle\tau\rangle \;&=\; (\Lambda(\alpha \geqslant \bot)\ a)\,(\forall\,(\geqslant\tau); \&) && (1)\\
&\longrightarrow (\Lambda(\alpha \geqslant \bot)\ a)\,(\forall\,(\geqslant\tau))\,(\&) && (2)\\
&\longrightarrow (\Lambda(\alpha \geqslant \bot\tau)\ a\{!\alpha \leftarrow \tau; !\alpha\})\,(\&) \;=\; (\Lambda(\alpha \geqslant \tau)\ a)\,(\&) && (3)\\
&\longrightarrow a\{!\alpha \leftarrow \mathbb{1}\}\{\alpha \leftarrow \tau\} \;=\; a\{\alpha \leftarrow \tau\} && (4)
\end{aligned}$$

Step (1) is by definition; step (2) is by ι-Seq; step (3) is by ι-Inside, step (4) is by ι-Elim and equality steps (3) and (4) are by type instantiation and by assumption as a is a term of System F, thus in which $!\alpha$ does not appear.

2 Properties of reduction

The reduction has been defined so that the type erasure of a reduction sequence in $x\mathsf{ML}^{\mathsf{F}}$ is a reduction sequence in the untyped λ-calculus. Formally, the type

erasure of a term a of $x\mathsf{ML}^\mathsf{F}$ is the untyped λ-term $\lceil a \rceil$ defined inductively by

$$\begin{array}{rlrl} \lceil x \rceil &= x & \lceil \mathsf{let}\ x = a_1\ \mathsf{in}\ a_2 \rceil &= \mathsf{let}\ x = \lceil a_1 \rceil\ \mathsf{in}\ \lceil a_2 \rceil \\ \lceil a\,\phi \rceil &= \lceil a \rceil & \lceil \lambda(x:\tau)\ a \rceil &= \lambda(x)\ \lceil a \rceil \\ \lceil a_1\ a_2 \rceil &= \lceil a_1 \rceil\ \lceil a_2 \rceil & \lceil \Lambda(\alpha \geqslant \tau)\ a \rceil &= \lceil a \rceil \end{array}$$

It is immediate to verify that two terms related by ι-reduction have the same type erasure. Moreover, if a β-reduces to a', then the type erasure of a β-reduces to the type erasure of a' in one step in the untyped λ-calculus.

2.1 Subject reduction

Reduction of $x\mathsf{ML}^\mathsf{F}$, which can occur in any context, preserves typings. This relies on weakening and substitution lemmas for both instance and typing judgments.

Lemma 3 (Weakening). *Let $\Gamma, \Gamma', \Gamma''$ be a well-formed environment.*
If $\Gamma, \Gamma'' \vdash \phi : \tau_1 \leq \tau_2$, then $\Gamma, \Gamma', \Gamma'' \vdash \phi : \tau_1 \leq \tau_2$.
If $\Gamma, \Gamma'' \vdash a : \tau'$, then $\Gamma, \Gamma', \Gamma'' \vdash a : \tau'$.

Lemma 4 (Term substitution). *Assume that $\Gamma \vdash a' : \tau'$ holds.*
If $\Gamma, x : \tau', \Gamma' \vdash \phi : \tau_1 \leq \tau_2$ then $\Gamma, \Gamma' \vdash \phi : \tau_1 \leq \tau_2$.
If $\Gamma, x : \tau', \Gamma' \vdash a : \tau$, then $\Gamma, \Gamma' \vdash a\{x \leftarrow a'\} : \tau$

The next lemma, which expresses that we can substitute an instance bound inside judgments, ensures the correctness of Rule ι-Elim.

Lemma 5 (Bound substitution). *Let φ and θ be respectively the substitutions $\{\alpha \leftarrow \tau\}$ and $\{!\alpha \leftarrow \mathbb{1}\}\{\alpha \leftarrow \tau\}$.*
If $\Gamma, \alpha \geqslant \tau, \Gamma' \vdash \phi : \tau_1 \leq \tau_2$ then $\Gamma, \Gamma'\varphi \vdash \phi\theta : \tau_1\varphi \leq \tau_2\varphi$.
If $\Gamma, \alpha \geqslant \tau, \Gamma' \vdash a : \tau'$ then $\Gamma, \Gamma'\varphi \vdash a\theta : \tau'\varphi$.

Finally, the following lemma ensures that an instance bound can be instantiated, proving in turn the correctness of the rule ι-Inside.

Lemma 6 (Narrowing). *Assume $\Gamma \vdash \phi : \tau \leq \tau'$. Let θ be $\{!\alpha \leftarrow \phi; !\alpha\}$.*
If $\Gamma, \alpha \geqslant \tau, \Gamma' \vdash \phi' : \tau_1 \leq \tau_2$ then $\Gamma, \alpha \geqslant \tau', \Gamma' \vdash \phi'\theta : \tau_1 \leq \tau_2$.
If $\Gamma, \alpha \geqslant \tau, \Gamma' \vdash a : \tau''$ then $\Gamma, \alpha \geqslant \tau', \Gamma' \vdash a\theta : \tau''$

Subject reduction is an easy consequence of all these results.

Theorem 1 (Subject reduction). *If $\Gamma \vdash a : \tau$ and $a \longrightarrow a'$ then, $\Gamma \vdash a' : \tau$.*

2.2 Confluence

Theorem 2. *The relation $\longrightarrow_\beta$ is confluent. The relations $\longrightarrow_\iota$ and $\longrightarrow$ are confluent on the terms well-typed in some context.*

This result is proved using the standard technique of parallel reductions (Barendregt 1984). Thus β-reduction and ι-reduction are independent; this allows for instance to perform ι-reductions under λ-abstractions as far as possible while keeping a weak evaluation strategy for β-reduction.

The restriction to well-typed terms for the confluence of ι-reduction is due to two things. First, the rule ι-Inside is not applicable to ill-typed terms in which $\tau\,\phi$ cannot be computed (for example $(\Lambda(\alpha \geqslant \mathsf{int})\ a)\,(\forall\,(\geqslant \&))$). Second, $\tau\,\phi$ can sometimes be computed, even though $\Gamma \vdash \phi : \tau \leq \tau'$ never holds (for example if ϕ is $!\alpha$ and τ is not the bound of α in Γ). Hence, type errors may be either revealed or silently reduced and perhaps eliminated, depending on the reduction path. As an example, let a be the term

$$\big(\Lambda(\alpha \geqslant \forall\,(\gamma)\ \gamma)\ \big((\Lambda(\beta \geqslant \mathsf{int})\ x)\,(\forall\,(\geqslant !\alpha))\big)\big)\,(\forall\,(\geqslant \&))$$

We have both $a \longrightarrow \Lambda(\alpha \geqslant \bot)\ \big((\Lambda(\beta \geqslant \mathsf{int})\ x)\,(\forall\,(\geqslant \&; !\alpha))\big) \not\longrightarrow$, and $a \longrightarrow \big(\Lambda(\alpha \geqslant \forall\,(\gamma)\ \gamma)\ \Lambda(\beta \geqslant \alpha)\ x\big)\,(\forall\,(\geqslant \&)) \longrightarrow \Lambda(\alpha \geqslant \bot)\ \Lambda(\beta \geqslant \alpha)\ x \not\longrightarrow$.

The fact that ill-typed terms may not be confluent is not new: for instance, this is already the case with η-reduction in System F. We believe this is not a serious issue. In practice, this means that typechecking should be performed before any program simplification, which is usually the case anyway.

2.3 Strong normalization

We conjecture, but have not proved, that all reduction sequences are finite.

2.4 Accommodating weak reduction strategies and constants

In order to show that the calculus may also be used as the core of a programming language, we now introduce constants and restricts the semantics to a weak evaluation strategy.

We let the letter c range over constants. Each constant comes with its arity $|c|$. The dynamic semantics of constants must be provided by primitive reduction rules, called δ-rules. However, these are usually of a certain form. To characterize δ-rules (and values), we partition constants into *constructors* and *primitives*, ranged over by letters C and f, respectively. The difference between the two lies in their semantics: primitives (such as $+$) are reduced when fully applied, while constructors (such as `cons`) are irreducible and typically eliminated when passed as argument to primitives.

In order to classify constructed values, we assume given a collection of type constructors κ, together with their arities $|\kappa|$. We extend types with constructed types $\kappa\ (\tau_1, \ldots \tau_{|\kappa|})$. We write $\overline{\alpha}$ for a sequence of variables $\alpha_1, \ldots \alpha_k$ and $\forall\,(\overline{\alpha})\ \tau$ for the type $\forall\,(\alpha_1)\ \ldots \forall\,(\alpha_k)\ \tau$. The static semantics of constants is given by an initial typing environment Γ_0 that assigns to every constant c a type τ of the form $\forall\,(\overline{\alpha})\ \tau_1 \to \ldots \tau_n \to \tau_0$, where τ_0 is a constructed type whenever the constant c is a constructor.

We distinguish a subset of terms, called values and written v. Values are term abstractions, type abstractions, full or partial applications of constructors, or partial applications of primitives. We use an auxiliary letter w to characterize the arguments of functions, which differ for call-by-value and call-by-name strategies. In values, an application of a constant c can involve a series of type instantiations, but only evaluated ones and placed before all other arguments.

Moreover, the application may only be partial whenever c is a primitive. Evaluated instantiations θ may be quantifier eliminations or either inside or under (general) instantiations. In particular, $a\,\tau$ and $a\,(!\alpha)$ are *never* values. The grammar for values and evaluated instantiations is as follows:

$$\begin{array}{rcll} v & ::= & \lambda(x:\tau)\ a & \\ & \mid & \Lambda(\alpha:\tau)\ a & \\ & \mid & C\,\theta_1 \ldots \theta_k\ w_1 \ldots w_n & n \leq |C| \\ & \mid & f\ \theta_1 \ldots \theta_k\ w_1 \ldots w_n & n < |f| \\ \theta & ::= & \forall\,(\geqslant \phi)\ \mid\ \forall\,(\alpha \geqslant)\ \phi\ \mid\ \& & \end{array}$$

Finally, we assume that δ-rules are of the form $f\ \theta_1 \ldots \theta_k\ w_1 \ldots w_{|f|} \longrightarrow_f a$ (that is, δ-rules may only reduce fully applied primitives).

In addition to this general setting, we make further assumptions to relate the static and dynamic semantics of constants.

Subject reduction: δ-reduction preserves typings, *i.e.*, for any typing context Γ such that $\Gamma \vdash a : \tau$ and $a \longrightarrow_f a'$, the judgment $\Gamma \vdash a' : \tau$ holds.

Progress: Well-typed, full applications of primitives can be reduced, *i.e.*, for any term a of the form $f\ \ \theta_1\ \ \ldots\ \ \theta_k\ w_1 \ldots w_n$ verifying $\Gamma_0 \vdash a : \tau$, there exists a term a' such that $a \longrightarrow_f a'$.

Call-by-value reduction We now specialize the previous framework to a call-by-value semantics. In this case, arguments of applications in values are themselves restricted to values, *i.e.* w is taken equal to v. Rules (β) and (β_{let}) are limited to the substitution of values, that is, to reductions of the form $(\lambda(x : \tau)\ a)\ v \longrightarrow a\{x \leftarrow v\}$ and $\mathsf{let}\ x = v\ \mathsf{in}\ a \longrightarrow a\{x \leftarrow v\}$. Rules ι-Id, ι-Comp and ι-Intro are also restricted so that they only apply to values (*e.g.* a is textually replaced by v in each of these rules). Finally, we restrict rule Context to call-by-value contexts, which are of the form

$$E_v\ ::=\ [\,\cdot\,]\ \mid\ E_v\ a\ \mid\ v\ E_v\ \mid\ E_v\ \phi\ \mid\ \mathsf{let}\ x = E_v\ \mathsf{in}\ a$$

We write $\longrightarrow\!\!\!\!\rightarrow_{\mathsf{v}}$ the resulting reduction relation. It follows from the above restrictions that the reduction is deterministic. Moreover, since δ-reduction preserves typings, by asumption, the relation $\longrightarrow\!\!\!\!\rightarrow_{\mathsf{v}}$ also preserves typings by Theorem 1.

Progress holds for call-by-value. In combination with subject-reduction, this ensures that the evaluation of well-typed terms "cannot go wrong".

Theorem 3. *If $\Gamma_0 \vdash a : \tau$, then either a is a value or $a \longrightarrow\!\!\!\!\rightarrow_{\mathsf{v}} a'$ for some a'.*

Call-by-value reduction and the value restriction The value-restriction is the most standard way to add side effects in a call-by-value language. It is thus important to verify that it can be transposed to $x\mathsf{ML}^\mathsf{F}$.

Typically, the *value restriction* amounts to restricting type generalization to non-expansive expressions, which contain at least value-forms, *i.e.* values and

term variables, as well as their type-instantiations. Hence, we obtain the following revised grammar for expansive expressions b and for non-expansive expressions u.

$$\begin{array}{rcll} b & ::= & u \mid b\,b \mid \mathsf{let}\ x = u\ \mathsf{in}\ b & \\ u & ::= & x \mid \lambda(x:\tau)\ b \mid \Lambda(\alpha:\tau)\ u \mid u\,\phi & \\ & \mid & C\,\theta_1 \ldots \theta_k\,u_1 \ldots u_n & n \leq |C| \\ & \mid & f\,\theta_1 \ldots \theta_k\,u_1 \ldots u_n & n < |f| \end{array}$$

As usual, we restrict let-bound expressions to be non-expansive, since they implicitly contain a type generalization. Notice that, although type instantiations are restricted to non-expansive expressions, this is not a limitation: $b\,\phi$ can always be written as $(\lambda(x:\tau)\ x\,\phi)\ b$, where τ is the type of a, and similarly for applications of constants to expansive expressions.

Theorem 4. *Expansive and non-expansive expressions are closed by call-by-value reduction.*

Corollary 1. *Subject reduction holds with the value restriction.*

It is then routine work to extend the semantics with a global store to model side effects and verify type soundness for this extension.

Call-by-name reduction For call-by-name reduction semantics, we can actually increase the set of values, which may contain applications of constants to arbitrary expressions; that is, we take a for w. The ι-reduction is restricted as for call-by-value. However, evaluation contexts are now $E_n ::= [\,\cdot\,] \mid E_n\ a \mid E_n\,\phi$. We write $\longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{n}}$ the resulting reduction relation. As for call-by-value, it is deterministic by definition and preserves typings. It may also always progress.

Theorem 5. *If $\Gamma_0 \vdash a : \tau$, then either a is a value or $a \longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{n}} a'$ for some a'.*

3 Discussion

Elaboration of graphical $e\mathsf{ML}^\mathsf{F}$ into $x\mathsf{ML}^\mathsf{F}$ To verify that, as expected, $x\mathsf{ML}^\mathsf{F}$ can be used as an internal language for $e\mathsf{ML}^\mathsf{F}$, we have exhibited a type-preserving type-erasure-preserving translation from $e\mathsf{ML}^\mathsf{F}$ to $x\mathsf{ML}^\mathsf{F}$. Technically, this translation is based on the presolutions of type inference problems in the graphic constraint framework of ML^F. An important corollary is the type soundness of $e\mathsf{ML}^\mathsf{F}$—in its most expressive[6] version (Rémy and Yakobowski 2008b). By lack of space, this translation is however not presented in this paper, but can be found in (Rémy and Yakobowski 2008a). We also expect that $x\mathsf{ML}^\mathsf{F}$ could be used as an internal language for HML, another less expressive but simpler surface language for $i\mathsf{ML}^\mathsf{F}$ that has been recently proposed (Leijen 2009).

[6] So far, type-soundness has only been proved for the original, but slightly weaker variant of ML^F (Le Botlan 2004) and for the shallow, recast version of ML^F (Le Botlan and Rémy 2007).

Expressiveness of $x\mathsf{ML}^\mathsf{F}$ The translation of $e\mathsf{ML}^\mathsf{F}$ into $x\mathsf{ML}^\mathsf{F}$ shows that $x\mathsf{ML}^\mathsf{F}$ is at least as expressive as $e\mathsf{ML}^\mathsf{F}$. However, and perhaps surprisingly, the converse is not true. That is, there exist programs of $x\mathsf{ML}^\mathsf{F}$ that cannot be typed in ML^F. While, this is mostly irrelevant when using ML^F as an internal language for $e\mathsf{ML}^\mathsf{F}$, the question is still interesting from a theoretical point of view, as understanding $x\mathsf{ML}^\mathsf{F}$ on its own, *i.e.* independently of the type inference constraints of $e\mathsf{ML}^\mathsf{F}$, could perhaps suggest other useful extensions of $x\mathsf{ML}^\mathsf{F}$.

For the sake of simplicity, we explain the difference between $x\mathsf{ML}^\mathsf{F}$ and $i\mathsf{ML}^\mathsf{F}$, the Curry-style version of ML^F (which has the same expressiveness as $e\mathsf{ML}^\mathsf{F}$). Although syntactically identical, the types of $x\mathsf{ML}^\mathsf{F}$ and of syntactic $i\mathsf{ML}^\mathsf{F}$ differ in their interpretation of alias bounds, *i.e.* quantifications of the form $\forall\,(\beta \geqslant \alpha)\ \tau$. Consider, for example, the two types τ_0 and τ_{id} defined as $\forall\,(\alpha \geqslant \tau)\ \forall\,(\beta \geqslant \alpha)\ \beta \to \alpha$ and $\forall\,(\alpha \geqslant \tau)\ \alpha \to \alpha$. In $i\mathsf{ML}^\mathsf{F}$, alias bounds can be expanded and τ_0 and τ_{id} are equivalent. Roughly, the set of their instances (stripped of toplevel quantifiers) is $\{\tau' \to \tau' \mid \tau \leq \tau'\}$. In contrast, the set of instances of τ_0 is larger in $x\mathsf{ML}^\mathsf{F}$ and at least a superset of $\{\tau'' \to \tau' \mid \tau \leq \tau' \leq \tau''\}$. This level of generality cannot be expressed in $i\mathsf{ML}^\mathsf{F}$.

The current treatment of alias bounds in $x\mathsf{ML}^\mathsf{F}$ is quite natural in a Church-style presentation. Surprisingly, it is also simpler than treating them as in $e\mathsf{ML}^\mathsf{F}$. A restriction of $x\mathsf{ML}^\mathsf{F}$ without alias bounds that is closed under reduction and in closer correspondence with $i\mathsf{ML}^\mathsf{F}$ can still be defined a posteriori, by constraining the formation of terms, but the definition is contrived and unnatural. Instead of restricting $x\mathsf{ML}^\mathsf{F}$ to match the expressiveness of $i\mathsf{ML}^\mathsf{F}$, a question worth further investigation is whether the treatment of alias bounds could be enhanced in $i\mathsf{ML}^\mathsf{F}$ and $e\mathsf{ML}^\mathsf{F}$ to match the one in $x\mathsf{ML}^\mathsf{F}$ without compromising type inference.

Related works A strong difference between $e\mathsf{ML}^\mathsf{F}$ and $x\mathsf{ML}^\mathsf{F}$ is the use of explicit coercions to trace the derivation of type instantiation judgments. A similar approach has already been used in a language with subtyping and intersection types, proposed as a target for the compilation of bounded polymorphism (Crary 2000). In both cases, coercions are used to make typechecking a trivial process. In our case, they are also exploited to make subject reduction easy—by introducing the language to describe how type instance derivations must be transformed during reduction. (We believe that the use of explicit coercions for simplifying subject-reduction proofs has been neglected.) In both approaches, reduction is split into a standard notion of β-reduction and a new form of reduction (which we call ι-reduction) that only deals with coercions, preserves type-erasures, and is (conjectured to be) strongly normalizing. There are also important differences. While both coercion languages have common forms, our coercions intendedly keep the instance-bounded polymorphism form $\forall\,(\alpha \geqslant \tau)\ \tau'$. On the opposite, coercions are used to eliminate the subtype-bounded polymorphism form $\forall\,(\alpha \leq \tau)\ \tau'$ in (Crary 2000), using intersection types and contravariant arrow coercions instead, which we do not need. It would be worth checking whether union types, which are propsoed as an extension in (Crary 2000), could be used to encode away our instance-bounded polymorphism form.

Besides this work and the several papers that describe variants of ML^F, there are actually few other related works. Both Leijen and Löh (2005) and Leijen (2007) have studied the extension of ML^F with qualified types, and as a subcase, the translation of ML^F without qualified types into System F. However, in order to handle type instantiations, a term a of type $\forall\,(\alpha \geqslant \tau')\ \tau$ is elaborated as a function of type $\forall\,(\alpha)\ (\tau'_\star \to \alpha) \to \tau_\star$, where $\tau_\star$ is a runtime representation of τ. The first argument is a *runtime coercion*, which bears strong similarities with our instantiations. However, an important difference is that their coercions are at the level of terms, while our instantiations are at the level of types. In particular, although coercion functions should not change the semantics, this critical result has not been proved so far, while in our settings the type-erasure semantics comes for free by construction. The impact of coercion functions in a call-by-value language with side effects is also unclear. Perhaps, a closer connection between their coercion functions and our instantiations could be established and used to actually prove that their coercions do not alter the semantics. However, even if such a result could be proved, coercions should preferably remain at the type level, as in our setting, than be intermixed with terms, as in their proposal.

Future works The demand for an internal language for ML^F was first made in the context of using the $e\mathsf{ML}^\mathsf{F}$ type system for the Haskell language. We expect $x\mathsf{ML}^\mathsf{F}$ to better accommodate qualified types than $e\mathsf{ML}^\mathsf{F}$ since no evidence function would be needed for flexible polymorphism, but it remains to be verified.

Type instantiation, which changes the type of an expression without changing its meaning, goes far beyond type application in System F and resembles retyping functions in System F^η—the closure of F by η-conversion (Mitchell 1988). Those functions can be seen either at the level of terms, as expressions of System F that $\beta\eta$-reduces to the identity, or at the level of types as a *type conversion*. Some loose parallel can be made between the encoding of ML^F in System F by Leijen and Löh (2005) which uses term-level coercions, and $x\mathsf{ML}^\mathsf{F}$ which uses type-level instantiations. Additionally, perhaps F^η could be extended with a form of abstraction over retyping functions, much as type abstraction $\forall\,(\alpha \geqslant \tau)$ in $x\mathsf{ML}^\mathsf{F}$ amounts to abstracting over the instantiation $!\alpha$ of type $\tau \to \alpha$. (Or perhaps, as suggested by the work of Crary (2000), intersection and union types could be added to F^η to avoid the need for abstracting over coercion functions.)

Regarding type soundness, it is also worth noticing that the proof of subject reduction in $x\mathsf{ML}^\mathsf{F}$ does not subsume, but complements, the one in the original presentation of ML^F. The latter does not explain how to transform type annotations, but shows that annotation sites need not be introduced (only transformed) during reduction. Because $x\mathsf{ML}^\mathsf{F}$ has full type information, it cannot say anything about type information that could be left implicit and inferred. Given a term in $x\mathsf{ML}^\mathsf{F}$, can we rebuild a term in $i\mathsf{ML}^\mathsf{F}$ with minimal type annotations? While this should be easy if we require that corresponding subterms have identical types in $x\mathsf{ML}^\mathsf{F}$ and $i\mathsf{ML}^\mathsf{F}$, the answer is unclear if we allow subterms to have different types.

The semantics of $x\mathsf{ML}^\mathsf{F}$ allows reduction (and elimination) of type instantiations $a\,\phi$ through ι-reduction but does not operate reduction (and simplification) of instantiations ϕ alone. It would be possible to define a notion of reduction on instantiations $\phi \longrightarrow \phi'$ (such that, for instance, $\forall\,(\geqslant \phi_1; \phi_2) \longrightarrow \forall\,(\geqslant \phi_1); \forall\,(\geqslant \phi_2)$, or conversely?) and extend the reduction of terms with a context rule $a\,\phi \longrightarrow a\,\phi'$ whenever $\phi \longrightarrow \phi'$. This might be interesting for more economical representations of instantiation. However, it is unclear whether there exists an interesting form of reduction that is both Church-Rosser and large enough for optimization purposes. Perhaps, one should rather consider instantiation transformations that preserve observational equivalence, which would leave more freedom in the way one instantiation could be replaced by another.

Extending $x\mathsf{ML}^\mathsf{F}$ to allow higher-order polymorphism is another interesting research direction for the future. Such an extension is already under investigation for the type inference version $e\mathsf{ML}^\mathsf{F}$ (Herms 2009).

Conclusion We have completed the ML^F trilogy by introducing the Church-style version $x\mathsf{ML}^\mathsf{F}$, that was still desperately missing for type-aware compilation and from a theoretical point of view. The original type-inference version $e\mathsf{ML}^\mathsf{F}$, which requires partial type annotations but does not tell how to track them during reduction, now lies between the Curry-style presentation $i\mathsf{ML}^\mathsf{F}$ that ignores all type information and $x\mathsf{ML}^\mathsf{F}$ that maintains it during reduction. We have shown that $x\mathsf{ML}^\mathsf{F}$ is well-behaved: reduction preserves well-typedness, and the calculus is sound for both call-by-value and call-by-name semantics.

Hence, $x\mathsf{ML}^\mathsf{F}$ can be used as an internal language for ML^F, with either call-by-value or call-by-name semantics, and also for the many restrictions of ML^F that have been proposed, including HML. Indeed, the translation of partially typed $e\mathsf{ML}^\mathsf{F}$ programs into fully typed $x\mathsf{ML}^\mathsf{F}$ ones, presented in (Rémy and Yakobowski 2008a), preserves well-typedness and the type erasure of terms, and therefore ensures the type soundness of $e\mathsf{ML}^\mathsf{F}$. Hopefully, this will help the adoption of ML^F and maintain a powerful form of type inference in modern programming languages that will necessarily feature first-class polymorphism.

Independently, the idea of enriching type applications to richer forms of type transformations might also be useful in other contexts.

References

Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984. ISBN 0-444-86748-1.

Karl Crary. Typed compilation of inclusive subtyping. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 68–81, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. doi: http://doi.acm.org/10.1145/351240.351247.

Paolo Herms. Partial Type Inference with Higher-Order Types. Master's thesis, University of Pisa and INRIA, 2009. To appear.

Didier Le Botlan. *MLF : An extension of ML with second-order polymorphism and implicit instantiation.* PhD thesis, Ecole Polytechnique, June 2004. english version.

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.

Didier Le Botlan and Didier Rémy. Recasting MLF. Research Report 6228, INRIA, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, June 2007.

Daan Leijen. A type directed translation of MLF to System F. In *The International Conference on Functional Programming (ICFP'07)*. ACM Press, October 2007.

Daan Leijen. Flexible types: robust type inference for first-class polymorphism. In *Proceedings of the 36th annual ACM Symposium on Principles of Programming Languages (POPL'09)*, pages 66–77, New York, NY, USA, 2009. ACM.

Daan Leijen and Andres Löh. Qualified types for MLF. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 144–155, New York, NY, USA, September 2005. ACM Press. ISBN 1-59593-064-7.

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

Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, May 2003. ISBN 0521826144.

Didier Rémy and Boris Yakobowski. A church-style intermediate language for MLF (extended version). Available at `http://gallium.inria.fr/~remy/mlf/xmlf.pdf`, September 2008a.

Didier Rémy and Boris Yakobowski. From ML to MLF: Graphic type constraints with efficient type inference. In *The 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 63–74, Victoria, BC, Canada, September 2008b.

Boris Yakobowski. *Graphical types and constraints: second-order polymorphism and inference.* PhD thesis, University of Paris 7, December 2008.

A Appendix: definition of $a\{!\alpha_0 \leftarrow \phi_0\}$

Formally, $a\,\theta$ where θ is $\{!\alpha_0 \leftarrow \phi_0\}$ is defined recursively as follows (we assume $\alpha \neq \alpha_0$). The interesting lines are the two first ones of the second column; other lines are just lifting the substitution from the leaves to types, type instantiations,

and terms in the usual way.

Types

$$\tau\,\theta = \tau$$

Terms

$$\begin{aligned} x\,\theta &= x \\ (a_1\ a_1)\,\theta &= (a_1\,\theta)\ (a_1\,\theta) \\ (a\,\phi)\,\theta &= a\,(\phi\,\theta) \\ (\lambda(x:\tau)\ a)\,\theta &= \lambda(x:\tau\,\theta)\ a\,\theta \\ (\Lambda(\alpha \geqslant \tau)\ a)\,\theta &= \Lambda(\alpha:\tau\,\theta)\ a\,\theta \end{aligned}$$

Type instantiations

$$\begin{aligned} !\alpha\,\theta &= !\alpha \\ !\alpha_0\,\theta &= \phi_0 \\ (\tau)\,\theta &= (\tau\,\theta) \\ (\forall\,(\geqslant \phi))\,\theta &= \forall\,(\geqslant \phi\,\theta) \\ (\forall\,(\alpha \geqslant)\ \phi)\,\theta &= \forall\,(\alpha \geqslant)\ (\phi\,\theta) \\ (\phi;\phi')\,\theta &= (\phi\,\theta);(\phi'\,\theta) \\ \&\,\theta &= \& \\ \parr\,\theta &= \parr \\ \mathbb{1}\,\theta &= \mathbb{1} \end{aligned}$$