A Church-Style Intermediate Language for MLF

Didier Rémy   Boris Yakobowski

October 27, 2009

Abstract

$\mathsf{ML^F}$ is a type system that seamlessly merges ML-style implicit but second-class polymorphism with System F explicit first-class polymorphism. We present $x\mathsf{ML^F}$, a Church-style version of $\mathsf{ML^F}$ 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 $x\mathsf{ML^F}$ is more general than type application in System F. We equip $x\mathsf{ML^F}$ 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. We exhibit a type preserving encoding of $\mathsf{ML^F}$ into $x\mathsf{ML^F}$, which ensures type soundness for the most general version of $\mathsf{ML^F}$. We observe that $x\mathsf{ML^F}$ is a calculus of retyping functions at the type level.

Introduction

$\mathsf{ML^F}$ (Le Botlan and Rémy 2003, 2007; Rémy and Yakobowski 2008) 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, maybe surprisingly, 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 $\mathsf{ML^F}$ 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.

iML$^\mathsf{F}$ and eML$^\mathsf{F}$ Currently, the language ML$^\mathsf{F}$ comes with a Curry-style version iML$^\mathsf{F}$, where no type information is needed, and a type-inference version eML$^\mathsf{F}$, that requires partial type information (Le Botlan and Rémy 2007). However, eML$^\mathsf{F}$ is not quite in Church's style, since a large amount of type information is still inferred, and partial type information cannot be easily maintained during reduction. Hence, while eML$^\mathsf{F}$ is a good surface language, it is not a good candidate for use as an internal language during the compilation process, where some simple program transformations, and perhaps some reduction steps, are being performed. This has been a problem for the adoption of ML$^\mathsf{F}$ in the Haskell community (Peyton Jones 2003), as the Haskell compilation chain uses an internal explicitly typed language, especially, but not only, for evidence translation due to the use of qualified types (Jones 1994).

This is also an obstacle to proving subject reduction, which does not hold in eML$^\mathsf{F}$. 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 eML$^\mathsf{F}$ may be equipped with reduction rules for type annotations. This has actually been considered in the original presentation of ML$^\mathsf{F}$, 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 (2008) have introduced graphic constraints, both to simplify the presentation of ML$^\mathsf{F}$ and improve its type inference algorithm. This also lead to a simpler, slightly more expressive definition of ML$^\mathsf{F}$.

xML$^\mathsf{F}$ In this paper, we present a Church-style version of ML$^\mathsf{F}$, called xML$^\mathsf{F}$, which contains full type information. In fact, type checking becomes a simple and local verification process—by contrast with type inference in eML$^\mathsf{F}$, which is based on unification. In xML$^\mathsf{F}$, 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 actually proof evidences for the type instance relations in ML$^\mathsf{F}$.

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 $\mathsf{ML^F}$ for a call-by-name semantics for the first time. Notably, $x\mathsf{ML^F}$ is a conservative extension of System F.

To verify that, as expected, $x\mathsf{ML^F}$ can be used as an internal language for $e\mathsf{ML^F}$, we exhibit a type-preserving type-erasure-preserving translation from $e\mathsf{ML^F}$ to $x\mathsf{ML^F}$. This translation is based on typing derivations and thus performed after type inference. Technically, it is based on presolutions of type inference problems in the graphic constraint framework of $\mathsf{ML^F}$. An important corollary is the type soundness of $e\mathsf{ML^F}$—in its most expressive[1] version (Rémy and Yakobowski 2008). Therefore, $x\mathsf{ML^F}$ could also be used as an internal language for (and ensure the type soundness of) HML—another less expressive but simpler surface language for $i\mathsf{ML^F}$ that has been recently proposed (Leijen 2009).

Besides these practical issues, $x\mathsf{ML^F}$ might be interesting in its own right: type instantiations change the types of terms in ways that have some similarities, but also important differences, with retyping functions in the language F^η—the closure of F by η-expansion. In particular, type instantiations operate entirely at the level of types and not at the level of terms, hence, by construction, they do not carry any computational content.

Outline Perhaps surprisingly, but quite interestingly. the difficulty in defining an internal language for $\mathsf{ML^F}$ is not reflected in the internal language itself, which, we believe, remains simple and easy to understand, but rather in the translation from $e\mathsf{ML^F}$ to $x\mathsf{ML^F}$, which is complicated by many administrative details. Hence, we present $x\mathsf{ML^F}$ first and study its meta-theoretical properties independently of $e\mathsf{ML^F}$. More precisely, the paper is organized as follows. We present $x\mathsf{ML^F}$, its syntax and its static and dynamic semantics in §1. We study its main properties, including type soundness for different evaluations strategies in §2. The elaboration of $e\mathsf{ML^F}$ programs into $x\mathsf{ML^F}$ is addressed in §3. We discuss possible improvements and variations, as well as related and future works at the end of the paper §4. All proofs are omitted, but can be found in (Yakobowski 2008, Chapters 14 & 15).

1 The calculus

1.1 Types, instantiations, terms, and typing environments

All the syntactic definitions of $x\mathsf{ML^F}$ 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

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

$\alpha, \beta, \gamma, \delta$			**Type variables**
τ	$::=$		**Types**
	$\mid$	α	Type variable
	$\mid$	$\tau \to \tau$	Arrow type
	$\mid$	$\forall\,(\alpha \geqslant \tau)\ \tau$	Flexible quantification
	$\mid$	$\bot$	Bottom type
ϕ	$::=$		**Instantiations**
	$\mid$	τ	Bottom
	$\mid$	$!\alpha$	Abstract
	$\mid$	$\forall\,(\geqslant \phi)$	Inside
	$\mid$	$\forall\,(\alpha \geqslant)\ \phi$	Under
	$\mid$	$\&$	Quantifier elimination
	$\mid$	$⅋$	Quantifier introduction
	$\mid$	$\phi; \phi$	Composition
	$\mid$	$\mathbb{1}$	Identity
x, y, z			**Term variables**
a	$::=$		**Terms**
	$\mid$	x	Variable
	$\mid$	$\lambda(x : \tau)\ a$	Function
	$\mid$	$a\ a$	Application
	$\mid$	$\Lambda(\alpha \geqslant \tau)\ a$	Type abstraction
	$\mid$	$a\,\phi$	Type instantiation
	$\mid$	$\mathsf{let}\ x = a\ \mathsf{in}\ a$	Let-binding
Γ	$::=$		**Environments**
	$\mid$	$\varnothing$	Empty environment
	$\mid$	$\Gamma, \alpha \geqslant \tau$	Type variable
	$\mid$	$\Gamma, x : \tau$	Term variable

Figure 1: Grammar of types, instantiations, and terms

type $\forall\alpha.\alpha$. Finally, a type may also be a form of bounded quantification $\forall\,(\alpha \geqslant \tau)$ τ', called *flexible* quantification, that generalizes the $\forall\alpha.\tau$ form of System F. (We may simply write $\forall\,(\alpha)\ \tau'$ when the bound τ is $\bot$.) Intuitively, $\forall\,(\alpha \geqslant \tau)\ \tau'$ restricts the variable α to range only over instances of τ. The variable α is bound in τ' but not in τ.

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

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

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

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

Figure 2: Type instance

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 form $\Lambda\alpha.\, a$ of System F.

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 $\alpha \geqslant \tau$.

As usual, we assume that typing environments do not bind twice the same variable. We write $\mathsf{dom}(\Gamma)$ for the set of all term and type variables that are bound by Γ. All the free type variables appearing in a type of the environment Γ must be bound earlier in Γ. Formally, writing $\mathsf{ftv}(\tau)$ for the set of type variables that appear free in τ, the relation $\mathsf{ftv}(\tau) \subseteq \mathsf{dom}(\Gamma)$ must hold to form environments $\Gamma, \alpha \geqslant \tau, \Gamma'$ and $\Gamma, x : \tau, \Gamma'$. All environments in this paper implicitly verify both well-formedness hypotheses.

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 $s\{v \leftarrow s'\}$.

1.2 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.

$$
\begin{array}{lll}
\tau\,(!\alpha) = \alpha & \bot\,\tau \;=\; \tau & \tau\,\mathbb{1} = \tau \\
\end{array}
$$

$$
\begin{array}{rcll}
\tau\ \text{⅋} & = & \forall\,(\alpha \geqslant \bot)\ \tau & \alpha \notin \mathsf{ftv}(\tau) \\
\tau\ (\phi_1; \phi_2) & = & (\tau\,\phi_1)\,\phi_2 & \\
(\forall\,(\alpha \geqslant \tau)\ \tau')\ \& & = & \tau'\{\alpha \leftarrow \tau\} & \\
(\forall\,(\alpha \geqslant \tau)\ \tau')\ (\forall\,(\geqslant \phi)) & = & \forall\,(\alpha \geqslant \tau\,\phi)\ \tau' & \\
(\forall\,(\alpha \geqslant \tau)\ \tau')\ (\forall\,(\alpha \geqslant)\ \phi) & = & \forall\,(\alpha \geqslant \tau)\ (\tau'\,\phi) &
\end{array}
$$

Figure 3: Type instantiation (on types)

The *quantifier introduction* ⅋[2] 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.

Example Let τ_{min}, τ_{cmp}, and τ_{and} be the types (for example, of the parametric minimum and comparison functions and the conjunction of boolean formulas) defined as follows:

$$
\begin{array}{rcl}
\tau_{\mathsf{min}} & \triangleq & \forall\,(\alpha \geqslant \bot)\ \alpha \to \alpha \to \alpha \\
\tau_{\mathsf{cmp}} & \triangleq & \forall\,(\alpha \geqslant \bot)\ \alpha \to \alpha \to \mathsf{bool} \\
\tau_{\mathsf{and}} & \triangleq & \mathsf{bool} \to \mathsf{bool} \to \mathsf{bool}
\end{array}
$$

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[3] $\forall\,(\alpha \geqslant)\ (\forall\,(\geqslant \alpha); \&)$. 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* $\tau_1 = \tau_2$*.*

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

[3] Notice that the occurrence of α in the inside instantiation is bound by the under instantiation.

$$\frac{\textsc{Var}\quad x : \tau \in \Gamma}{\Gamma \vdash x : \tau}
\qquad
\frac{\textsc{Let}\quad \Gamma \vdash a : \tau \qquad \Gamma, x : \tau \vdash a' : \tau'}{\Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \tau'}
\qquad
\frac{\textsc{Abs}\quad \Gamma, x : \tau \vdash a : \tau'}{\Gamma \vdash \lambda(x : \tau)\ a : \tau \to \tau'}$$

$$\frac{\textsc{App}\quad \Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1}
\qquad
\frac{\textsc{TAbs}\quad \Gamma, \alpha \geqslant \tau' \vdash a : \tau \qquad \alpha \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \Lambda(\alpha \geqslant \tau')\ a : \forall\,(\alpha \geqslant \tau')\ \tau}$$

$$\frac{\textsc{TApp}\quad \Gamma \vdash a : \tau \qquad \Gamma \vdash \phi : \tau \leq \tau'}{\Gamma \vdash a\,\phi : \tau'}$$

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

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*[4], 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 $\Gamma \vdash \phi : \tau \leq \tau'$.

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 Section 4.1.

1.3 Typing rules for $x\mathsf{ML}^{\mathsf{F}}$

Typing rules are defined in Figure 4. Compared with System F, the novelties are, unsurprisingly, type abstraction and type instantiation. 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 to 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 $\tau'\{\alpha \leftarrow \tau\}$

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

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

Figure 5: Reduction rules

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 $\mathsf{choice\_id}\,(⅋; \forall\,(\gamma \geqslant)\ (\forall\,(\geqslant \langle\gamma\rangle); \&))$ has the ML type $\forall\,(\gamma \geqslant \bot)\ (\gamma \to \gamma) \to \gamma \to \gamma$.

1.4 Reduction

The semantics of the calculus is given by a small-step reduction semantics. We let reduction occur in any context, including under abstractions. That is, the evaluation contexts are all single-hole contexts, given by the grammar:

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

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[5] by the instantiation $(\phi; !\alpha)$. 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 §1.3 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 $\Lambda(\alpha \geqslant \bot)\ a$). In this case, all type instantiations are eliminated by reduction (but this not always the case in

[5]Here, the instantiation $!\alpha$ is seen as atomic.

general).

$$\begin{array}{lll}
\mathsf{choice\_id}\ \langle\langle \mathsf{int}\rangle\rangle & : & (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int}) \\
 & \longrightarrow\!\!\!\!\!\rightarrow & \lambda(y : \mathsf{int} \to \mathsf{int})\ (\lambda(x : \mathsf{int})\ x) \\[1ex]
\mathsf{choice\_id}\ \& & : & (\forall\,(\alpha)\ \alpha \to \alpha) \to (\forall\,(\alpha)\ \alpha \to \alpha) \\
 & \longrightarrow\!\!\!\!\!\rightarrow & \lambda(y : \forall\,(\alpha)\ \alpha \to \alpha)\ (\Lambda(\alpha)\ \lambda(x : \alpha)\ x) \\[1ex]
\multicolumn{3}{l}{\mathsf{choice\_id}\ (\parr; \forall\,(\gamma \geqslant)\ (\forall\,(\geqslant \langle\gamma\rangle); \&))} \\
 & : & \forall\,(\gamma)\ (\gamma \to \gamma) \to (\gamma \to \gamma) \\
 & \longrightarrow\!\!\!\!\!\rightarrow & \Lambda(\gamma)\ \lambda(y : \gamma \to \gamma)\ (\lambda(x : \gamma)\ x)
\end{array}$$

1.5 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:

$$\begin{aligned}
(\Lambda(\alpha)\ a)\ \langle\tau\rangle &= (\Lambda(\alpha \geqslant \bot)\ a)\ (\forall\,(\geqslant \tau); \&) && (1)\\
&\longrightarrow (\Lambda(\alpha \geqslant \bot)\ a)\ (\forall\,(\geqslant \tau))\ (\&) && (2)\\
&\longrightarrow (\Lambda(\alpha \geqslant \bot\tau)\ a\{!\alpha \leftarrow \tau; !\alpha\})\ (\&) && (3)\\
&= (\Lambda(\alpha \geqslant \tau)\ a)\ (\&) && (4)\\
&\longrightarrow a\{!\alpha \leftarrow \mathbb{1}\}\{\alpha \leftarrow \tau\} && (5)\\
&= a\{\alpha \leftarrow \tau\} && (6)
\end{aligned}$$

Step (1) is by definition; step (2) is by ι-Seq; step (3) is by ι-Inside, step (5) is by ι-Elim and steps (4) and (6) 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 (Barendregt 1984). 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

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

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

In this section, we show that reduction of $x\mathsf{ML}^{\mathsf{F}}$, which can occur in any context, preserves typings. As usual, this relies on weakening and substitution lemmas, which hold for both instance and typing judgments.

Lemma 3 (Weakening) *Assume that* $\Gamma, \Gamma', \Gamma''$ *is well-formed.*
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 that* $\Gamma \vdash \phi : \tau \leq \tau'$ *holds. 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

As expected, reduction is confluent.

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 $(\Lambda(\alpha \geqslant \mathsf{int})\; a)\; \&)$. 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, consider the term

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

It is ill-typed in any context, because $!\alpha$ coerces a term of type $\forall\,(\gamma)\;\gamma$ into one of type α, but $!\alpha$ is here indirectly applied to a term of type int. If we reduce the outermost type instantiation first, we are stuck with $\Lambda(\alpha \geqslant \bot)\;\big((\Lambda(\beta \geqslant \mathsf{int})\;x)\;(\forall\,(\geqslant \&;\,!\alpha))\big)$, which is irreducible since the type instantiation $\mathsf{int}\,(\&;\,!\alpha)$ is undefined.

Conversely, if we reduce the innermost type instantiation first, the faulty type instantiation disappears and we obtain the term $\big(\Lambda(\alpha \geqslant \forall\,(\gamma)\;\gamma)\;\Lambda(\beta \geqslant \alpha)\;x\big)\;(\forall\,(\geqslant \&))$, which further reduces to the normal form $\Lambda(\alpha \geqslant \bot)\;\Lambda(\beta \geqslant \alpha)\;x$.

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 checked, that all reduction sequences are finite in $x\mathsf{ML}^{\mathsf{F}}$.

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 will show that subject reduction and progress hold for the main two forms of weak-reduction strategies, namely call-by-value and call-by-name.

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 before all other arguments. Moreover, when c is a primitive the application may only be partial. 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:

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

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. That is, 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. That is, 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 is supposed to preserve typings, it is immediate by Theorem 1 that $\longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{v}}$ also preserves typings.

Crucially, 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 there exists a' such that $a \longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{v}} a'$.*

Call-by-value reduction and the value restriction The value-restriction (Wright and Felleisen 1994) 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.

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

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 now 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 of the grammatical form: $E_n ::= [\cdot] \mid E_n\; a \mid E_n\,\phi$. We write $\longrightarrow\!\!\!\!\!\rightarrow_{\mathsf{n}}$ the resulting reduction relation. As for call-by-value, it is deterministic by definition and it preserves typings. It may also always progress.

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

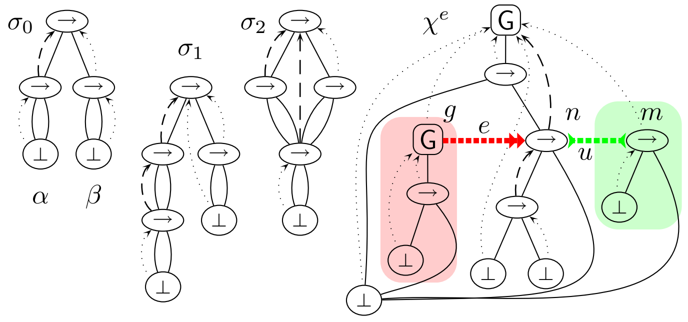


Figure 6: Types, constraints, and expansion

3 Elaboration of graphical eML$^{\mathsf{F}}$ into xML$^{\mathsf{F}}$

In this section, we study the translation of the graphical version of eML$^{\mathsf{F}}$ (Rémy and Yakobowski 2008; Yakobowski 2008) into xML$^{\mathsf{F}}$. The graphical version of eML$^{\mathsf{F}}$ is more general than the syntactic versions, and better suited for type inference; hence our choice. A full presentation of graphical eML$^{\mathsf{F}}$ is however out of the scope of this paper; we only remind the essential points in this section.

3.1 An overview of graphical eML$^{\mathsf{F}}$

Graphic types Types of graphical eML$^{\mathsf{F}}$ are graphs, designated with letter σ, composed of the superposition of a term-dag, representing the structure of the type, and of a binding tree encoding the binding information.

Term-dags are just dag representations of usual tree-like types, where at least all occurrences of the same variable must be shared, and inner nodes representing identical subtypes may also be shared. We write $\sigma(n)$ for the constructor at node n. Variables are anonymous, and represented by the pseudo-constructor $\bot$. Term-dag edges are written $n \overset{i}{\multimap} m$, where i is an integer that ranges between 1 and the arity of $\sigma(n)$; we also use the notation $\langle ni \rangle$ to designate m, the root node being simply noted $\langle\rangle$. In the drawings, edges are drawn with plain lines, oriented downwards. We leave i implicit, as outgoing edges are always drawn from left to right.

The binding tree is an upside-down tree with an edge $n \overset{\diamond}{\rightarrowtail} m$ leaving from each node n different from the root, and going to some node m (upper in the term-dag) at which n is bound. Binding edges may be either flexible or rigid, which is represented by labeling the edge with a flag $\diamond$ that is either $\geqslant$ or $=$, respectively. (On drawings, these flags are represented by dotted or dashed lines, respectively.)

Example Consider the graphic type σ_0 of Figure 6. The nodes $\langle 11 \rangle$ and $\langle 22 \rangle$ are variables (names α and β are here to help reading the figure, but they are not part of the graphic type). Paths 11 and 12 lead to the same node, which can therefore be called $\langle 11\rangle$ or $\langle 12\rangle$ indifferently. The edge $\langle 22\rangle \xrightarrow{\geq} \langle 2\rangle$ is a flexible binding edge (the rightmost lowermost one), while $\langle 1\rangle \xrightarrow{=} \langle\rangle$ is a rigid binding edge (the leftmost uppermost one) and $\langle 1\rangle \xrightarrow{2} \langle 12\rangle$ is a structure edge.

Binding edges express polymorphism. Typically, a rigid edge means that polymorphism is required, as for example the type of an argument that is used polymorphically. By contrast, a flexible edge means that polymorphism is available (as with flexible quantification in $x\mathsf{ML}^\mathsf{F}$) but not required. For example, σ_0 is the type of a function whose argument must be at least as polymorphic as $\forall(\alpha)\ \alpha \to \alpha$, and whose result has type $\forall(\beta)\ \beta \to \beta$, or any instance of it. In other words, if f is a function of type σ_0, the result of an application of f can be used in place of the successor function of type $\mathsf{int} \to \mathsf{int}$, but f cannot be passed the successor function as argument.

Rigid bounds arise from type annotations: in the absence of type annotations (and types with rigid bounds in the typing environment), polymorphism is offered, but is never requested, and the principal types of expressions only use flexible bounds.

For the purpose of defining type instance, we distinguish four kinds of nodes. Nodes on which no variable is transitively flexibly bound are called *inert*, as they neither hold nor control polymorphism. All other nodes hold or control polymorphism and are classified as follows. Nodes whose binding path is flexible up to the root are called *instantiable*; they can be freely instantiated as described next (in $x\mathsf{ML}^\mathsf{F}$ these nodes would correspond to parts of types that could be transformed by a suitable instantiation expression). Nodes whose binding edge is rigid are called *restricted*; they can only be transformed in a restricted way (in $x\mathsf{ML}^\mathsf{F}$ these nodes would correspond to polymorphic types occurring under some arrow type). Nodes whose binding edge is flexible but whose binding path up to the root contains a rigid edge are called *locked*; they cannot be transformed in any way (in $x\mathsf{ML}^\mathsf{F}$ these nodes would correspond to polymorphic types occurring in the bound of quantifiers themselves under some arrow type and not instantiation can transform them).

Type instance The *instance* relation on graphic types, written $\sqsubseteq$, is defined as the composition of four atomic operations: grafting, merging, raising and weakening. Grafting and merging are the usual instance transformations on first-order term-dags and do not change the binding tree. Conversely, weakening and raising only change the binding tree. Weakening transforms a flexible edge into a rigid one. Raising lets one binding edge slide over another one. Moreover, grafting is disallowed on restricted nodes and the four operations are disallowed on locked nodes.

Example (continued) In σ_1, the node $\langle 2\rangle$ is inert, $\langle 111\rangle$ is locked, $\langle 21\rangle$ is instantiable and $\langle 1\rangle$ is restricted. The graphic type σ_2 is an instance of σ_1, obtained by raising the node $\langle 11\rangle$, grafting then weakening $\langle 22\rangle$, and finally merging $\langle 11\rangle$ and $\langle 21\rangle$.

$$\llbracket\, y \,\rrbracket \;=^{(\mathbf{1})}\; y \qquad\qquad \llbracket\, z \,\rrbracket \;=^{(\mathbf{2})}\; \bigwedge\!\!\!\!\backslash(g)\; z\,(\Phi(e_z))$$

$$\llbracket\, \lambda(y)\; b \,\rrbracket \;=^{(\mathbf{3})}\; \bigwedge\!\!\!\!\backslash(g)\; \lambda(y : \mathcal{S}(y))\; \llbracket\, b \,\rrbracket\,(\Phi(e))$$

$$\llbracket\, b_1\; b_2 \,\rrbracket \;=^{(\mathbf{4})}\; \bigwedge\!\!\!\!\backslash(g)\; \llbracket\, b_1 \,\rrbracket\,(\Phi(e_1))\;\; (\llbracket\, b_2 \,\rrbracket\,(\Phi(e_2)))$$

$$\llbracket\, \mathsf{let}\; z = b_1 \;\mathsf{in}\; b_2 \,\rrbracket \;=^{(\mathbf{5})}\; \bigwedge\!\!\!\!\backslash(g)\; \mathsf{let}\; z = \llbracket\, b_1 \,\rrbracket \;\mathsf{in}\; \llbracket\, b_2 \,\rrbracket\,(\Phi(e_2))$$

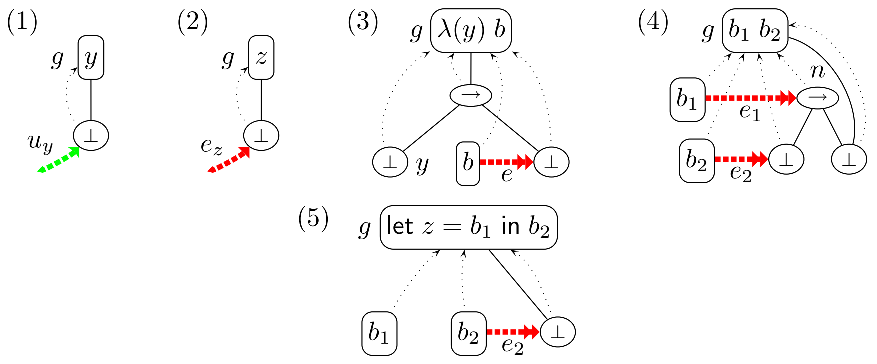


Figure 7: Constraint generation and translation of presolutions

Type constraints Type constraints generalize graphic types by adding new forms of edges, called constraint edges. These can be either *unification edges* $\rightarrowtail\!\leftarrowtail$ or *instantiation edges* $\twoheadrightarrow$. Instantiation edges are oriented. They relate special nodes, used to represent type schemes and called G-nodes, to regular nodes. An example of a constraint χ^e is shown on the right-hand side of Figure 6.

The instance on type constraints is exactly as on graphic types—constraint edges are just preserved.

A type constraint is solved when all of its constraint edges are solved. A unification edge is solved when it relates a node to itself (thus, a unification edge forces the nodes it relates to be merged). An instantiation edge e of the form $g \twoheadrightarrow n$ of a constraint χ is solved when, informally, n is an instance of the type scheme represented by g, or formally, when the expansion of e in χ is an instance of χ, as described below.

A solved instance of a constraint is called a *presolution*. It still contains all the nodes of the original constraint, many of which may have become irrelevant for describing the resulting type. A solution of a constraint is, roughly, a presolution in which such nodes have been removed. We need not define solutions formally since the translation uses presolutions directly.

Expansion Consider an instantiation edge e defined as $g \twoheadrightarrow n$ in a constraint χ. We define an *expansion* operation that enforces the constraint represented by e. The expansion of e in χ, written χ^e, is the constraint χ extended with both a copy of the type scheme represented by g and a unification edge between n and the root of the copy. The copy is bound at the same node as n.

Technically, we define the *interior* of g, written $\mathcal{I}(g)$ as all the nodes transitively bound to g. The expansion operation copies all the nodes structurally strictly under g and in the interior of g. Intuitively, those nodes are generic at the level of g. Conversely, the nodes under g that are not in the interior of g are not generic at the level of g and are not copied by the expansion (but are instead shared with the original).[6]

By construction, an instantiation edge e is solved if and only if χ is an instance of χ^e. We call *instantiation witness* an instance derivation of $\chi^e \sqsubseteq \chi$ for a solved instantiation edge e.

Example Let us consider the expansion χ^e of Figure 6. The original constraint χ can be obtained from χ^e by removing the rightmost highlighted nodes, as well as the resulting dangling edges. The interior of g is composed of the nodes in the leftmost box. Hence the copied nodes are $\langle g1 \rangle$ and $\langle g11 \rangle$, but *not* $\langle g12 \rangle$, which is not in $\mathcal{I}(g)$. The root of the expansion m is the copy of $\langle g1 \rangle$. It is bound to the binder of n and connected to n by the unification edge u.

In this example, χ is an instance of χ^e, as witnessed by the following operations: graft $\forall\,(\alpha)\,\forall\,(\beta)\;\alpha \to \beta$ under $\langle m1 \rangle$; raise $\langle m11 \rangle$ twice, and merge it with $\langle n11 \rangle$; weaken $\langle m1 \rangle$ and m; finally, merge n and m. Hence, the edge e (and χ itself) is solved.

From λ-terms to typing constraints Terms of $e\mathsf{ML}^{\mathsf{F}}$ are the partially annotated λ-terms generated by the following grammar

$$b ::= x \mid \lambda(x)\; b \mid \lambda(x : \sigma)\; b \mid b\; b \mid \mathsf{let}\; x = b\; \mathsf{in}\; b \mid (b : \sigma)$$

Source terms are translated into type constraints in a compositional manner. Every occurrence of a subexpression b is associated to a distinct G-node in the constraint, which we label with b for readability; however it should be understood that different occurrences of equal subexpressions b are mapped to different nodes. We let y and z stand for λ-bound and let-bound variables, respectively. Constraint generation is described on the bottom of Figure 7, for the expressions described by the left-hand sides of the equalities at the top of the figure. The unification edge u_y in (1) is linked to its corresponding variable node y generated in (3) by the translation of the abstraction binding y. The instantiation edge e_z in (2) is coming from the G-node labeled b_1 generated in (5) by the translation of the let expression binding z.

The constructions $\lambda(x : \sigma)\; b$ and $(b : \sigma)$ are actually syntactic sugar, for $\lambda(x)\; \mathsf{let}\; x = \kappa_\sigma\; x\; \mathsf{in}\; b$ and $\kappa_\sigma\; b$ respectively, where κ_σ is a coercion function. Both constructs are desugared before the translation into constraints.

Example The typing constraint χ for the term $\lambda(x)\; \lambda(y)\; x$ is described on the left-hand side of Figure 8. One of its presolutions χ_p is drawn on the middle

[6]Readers familiar with (Rémy and Yakobowski 2008) may notice a slight change in terminology, as in this work we use the term “expansion” instead of “propagation”, and we solve frontier unification edges on the fly.

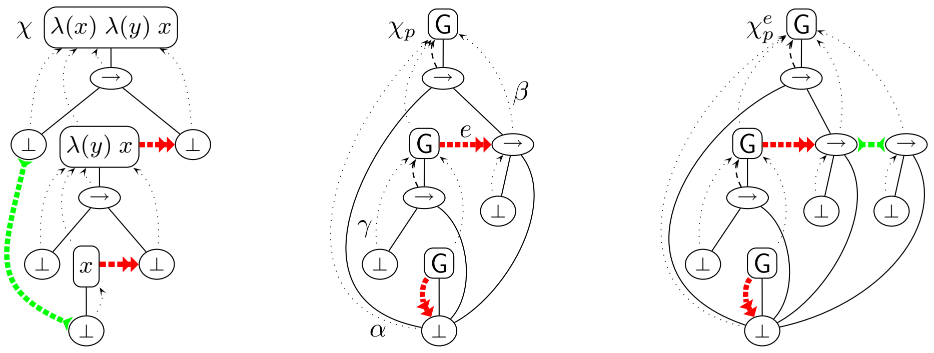


Figure 8: Typing constraints for $\lambda(x)\ \lambda(y)\ x$.

(We have dropped the mapping of expressions to G-node for conciseness, and labeled some binding edges that will appear in the xMLF translation.) Notice that this is not the most general presolution, as the arrow nodes bound at G-nodes have been made rigid, but an equivalent rigid presolution, as explained in §3.3, that is ready for translation into xMLF.

While type inference is out of the scope of this work—see (Rémy and Yakobowski 2008), we may however easily *check* that χ_p is indeed a presolution, *i.e.* that both instantiation edges are solved. Consider for example the edge e. We must verify that χ_p is an instance of the expansion χ_p^e drawn on the right-hand side, that is, exhibit a sequence of atomic instance operations that transforms χ_p^e into χ_p. Here, the obvious solution is just to merge the two nodes related by the unification edge.

3.2 An overview of the translation to xMLF

The translation of an eMLF term b to xMLF is based on a presolution χ of the typing constraint for b. Typing constraints have principal presolutions. However, any presolution—not merely the principal one, which is the one returned by type inference—can be translated. Since presolutions are instances of the original constraint, and type instance preserves both G-nodes and instantiation edges, we can refer to the original nodes and edges in Figure 7 when defining the translation. The translation is inductively defined on the structure of terms, reading auxiliary information on the corresponding nodes in the presolution to build the type of function parameters, type abstractions, and type instantiations. There are two key ingredients:

- For each instantiation edge e of the form $g \twoheadrightarrow n$, an instantiation $\Phi(e)$ is inserted to transform the type of the translation of the expression b corresponding to g into the type of n. It can be computed from the proof that e is solved in χ, *i.e.* from the instantiation witness for e. Details are given in §3.4.

- For each flexible binding $n \succ\!\!\longrightarrow g$, a type abstraction $\Lambda(\alpha_n \geqslant \tau_n)$ is inserted in front of the translation of the expression b corresponding to g, τ_n being the type of the node n. Indeed, such an edge corresponds to some polymorphism in n that must be introduced at the level of g. We use the notation $\bigwedge\!\!\!\!\!\setminus(g)$ to refer to all such quantifications at the level of g, which will be precisely defined in §3.4. (Rigid bindings, which are only useful to make type inference decidable, are inlined during the translation. Hence they do not give rise to type quantifications.)

The translation is given in Figure 7. When b is a λ-bound variable y (1), its translation is itself, as the G-node y is always monomorphic. For the other cases, the translation is of the form $\bigwedge\!\!\!\!\!\setminus(g)\ b'$, g being the G-node for b. Indeed, in ML^{F} and unlike in ML, generalization is as useful for applications and abstractions as for let-bound expressions. A variable z (2) bound by some $\mathsf{let}\ z = b_1\ \mathsf{in}\ b_2$ expression is instantiated by $\Phi(e_z)$ to transform the type of $[\![b_1]\!]$ into the type of this occurrence of z, according to the edge e_z. Correspondingly, in the translation of $\mathsf{let}\ z = b_1\ \mathsf{in}\ b_2$ (5), the translation of b_1 is bound to z uninstantiated, since each occurrence of z in $[\![b_2]\!]$ will potentially pick a different instance, while the translation of b_2 is instantiated according to the edge e_2. In the translation of an abstraction $\lambda(y)\ b$ (3), we annotate y by its type in the presolution (written $\mathcal{S}(y)$ and defined in §3.4) and coerce $[\![b]\!]$ to the its type inside the abstraction according to the edge e. Finally, the translation of an application (4) is the application of the translations, each of which is instantiated according to its constraint edge.

The translation is type-erasure preserving by construction.

Lemma 7 *Given a desugared term b, we have $\lceil [\![\, b \,]\!] \rceil = \lceil b \rceil$.*

Example The presolution χ_p in Figure 8 is translated to the term $\Lambda(\alpha)\ \Lambda(\beta \geqslant \forall\,(\delta)\ \delta \to \alpha)\ \lambda(x : \alpha)\ (\Lambda(\gamma)\ \lambda(y : \gamma)\ (x\ \mathbb{1}))\ (!\gamma)$ which has type $\forall\,(\alpha)\ \forall\,(\beta \geqslant \forall\,(\delta)\ \delta \to \alpha)\ \alpha \to \beta$. Notice the three type quantifications for α, β and γ that correspond to the flexible edges of the same name. The instantiation $!\gamma$ is the translation of e.

3.3 From presolutions to rigid presolutions

Some presolutions are not suited for translation, for two reasons.

Firstly, the following nodes, which may be flexibly bound to a G-node, must not result in a type quantification (as this would generate useless bindings, or even incorrect ones):

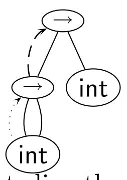
1. the node $\langle g1 \rangle$ in the translation of abstractions (3);
2. the node n in the translation of an application (4);
3. the node $\langle g1 \rangle$ whenever bound on g;

4. any node bound on a G-node but not reachable from a G-node by following only structure edges.

It is important that these nodes are retained and that their binding remain flexible *during* type inference when some of the constraints might not have yet been solved. However, their bindings may be made rigid *after* type inference, *i.e.* in presolutions, without actually loosing any expressiveness, as we shall see below. As a result, these nodes will be inlined during the translation into $x\mathsf{ML}^\mathsf{F}$.

Secondly, type equivalence in $e\mathsf{ML}^\mathsf{F}$ is larger than in $x\mathsf{ML}^\mathsf{F}$. Hence, some instance operations allowed in $e\mathsf{ML}^\mathsf{F}$ do not have a counterpart in $x\mathsf{ML}^\mathsf{F}$. In particular, $e\mathsf{ML}^\mathsf{F}$ allows instance operations on inert nodes. However, when the binding path of an inert node n contains a rigid binding, the translation of n into $x\mathsf{ML}^\mathsf{F}$ cannot be instantiated in $x\mathsf{ML}^\mathsf{F}$. Indeed, while type instantiation in $x\mathsf{ML}^\mathsf{F}$ can operate under flexible bounds using inside-instantiations, rigid nodes of $e\mathsf{ML}^\mathsf{F}$ are inlined and thus unreachable in $x\mathsf{ML}^\mathsf{F}$.

For example, the flexible binding edge in the type next, which is leaving from an inert node, may be weakened into $e\mathsf{ML}^\mathsf{F}$, leading to two equivalent types whose translations $(\forall\,(\alpha \geqslant \mathsf{int})\ \alpha \to \alpha) \to \mathsf{int}$ and $(\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$ are not equivalent in $x\mathsf{ML}^\mathsf{F}$ (and actually incomparable). Indeed, since type applications are explicit in $x\mathsf{ML}^\mathsf{F}$, a term of the former type must instantiate its argument before applying it, while a term of the latter type can apply its argument directly. This is quite similar to the difference between the two System F types $(\forall\,(\alpha)\ \mathsf{int} \to \mathsf{int}) \to \mathsf{int}$ and $(\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$.



The difference in type equivalence does not mean that $x\mathsf{ML}^\mathsf{F}$ is less expressive than $e\mathsf{ML}^\mathsf{F}$: inert nodes can always be weakened in presolutions of $e\mathsf{ML}^\mathsf{F}$. Moreover, we do not lose expressiveness by doing so, since this transformation commutes with other operations used to solve presolutions.

We call *rigid* a presolution that respects the conditions given at the beginning of this section and in which inert nodes are rigidly bound. The following lemma ensures that rigid-presolutions can be used in place of presolutions without affecting the set of solutions, up to weakening of inert nodes.

Lemma 8 *Given a presolution χ_p of a constraint χ, there exists a rigid presolution χ'_p of χ (derived from χ_p by weakening some nodes), in which terms have the same types as in χ modulo the weakening of inert nodes.*

This also suggests that we could have restricted ourselves to rigid presolution in the first place, since principal presolutions can be turned into rigid ones in a principal manner. However, rigid presolutions are only useful for the translation of $e\mathsf{ML}^\mathsf{F}$ into $x\mathsf{ML}^\mathsf{F}$ and useless, if not harmful, for type inference purposes: binding edges can only be rigidified—without loosing solutions—after all the constraint edges under them have been solved. This imposes synchronization in the constraint resolution. Therefore, we prefer to stay with the more flexible definition of presolutions for $e\mathsf{ML}^\mathsf{F}$ (thus avoiding unnecessary complications in the definition of $e\mathsf{ML}^\mathsf{F}$, which is exposed to the user) and only consider rigid presolution as a first step of the translation into $x\mathsf{ML}^\mathsf{F}$.

In the remainder of this section, we abstract over a rigid presolution χ and an instantiation edge e of the form $g \twoheadrightarrow d$.

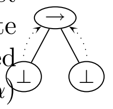
$$
\begin{array}{lcll}
\mathcal{T}(n) & \triangleq & \forall\,(\mathcal{Q}(n))\ \chi(n)\ (\mathcal{R}(\langle n\,1\rangle), \ldots \mathcal{R}(\langle n\,p\rangle)) & \\
 & & \multicolumn{2}{r}{\text{where } p \text{ is the arity of } \chi(n)} \\
\mathcal{R}(n) & \triangleq & \begin{cases} \mathcal{T}(n) & \text{if } n \text{ is rigid} \\ \alpha_n & \text{otherwise} \end{cases} & \\
\mathcal{Q}(n) & \triangleq & (\alpha_{\langle n_1\rangle} \geqslant \mathcal{T}(n_1) \ldots \alpha_{\langle n_k\rangle} \geqslant \mathcal{T}(n_k)) & \\
\multicolumn{4}{r}{\text{where } n_1, \ldots n_k \text{ are all nodes flexibly bound to } n, \text{ ordered}} \\
\mathcal{S}(n) & \triangleq & \begin{cases} \forall\,(\mathcal{Q}(n))\ \mathcal{S}(\langle n\,1\rangle) & \text{if } n \text{ is a } \mathsf{G}\text{-node} \\ \alpha_n & \text{if } n \text{ flexibly bound to a } \mathsf{G}\text{-node} \\ \mathcal{T}(n) & \text{otherwise} \end{cases} &
\end{array}
$$

Figure 9: Mapping nodes of $e\mathsf{ML}^{\mathsf{F}}$ to types of $e\mathsf{ML}^{\mathsf{F}}$

3.4 Details of the translation

Ordering binders and nodes In $e\mathsf{ML}^{\mathsf{F}}$, two binding edges reaching the same node are unordered. It is actually a useful property for type inference not to distinguish between two types that just differ by the order of their quantifiers. However, adjacent quantifiers do not commute in $x\mathsf{ML}^{\mathsf{F}}$. While they could be explicitly reordered by type instantiation, it is much better to get them in the right order by construction as far as possible (as described below however, reordering of quantifiers will still be necessary in some cases).

The simplest way to achieve this is to assume a total ordering $\prec$ of *all* nodes of χ. Of course, $\prec$ cannot be arbitrary, as it should also ensure the well-scopedness of syntactic types: if $n \multimap n'$ or $n' \rightarrowtail n$, then $n' \prec n$ must hold. We choose the leftmost-lowermost ordering of nodes for $\prec$: if $n_1, \ldots, n_k$ are bound to n, we first translate the n_i which is lowest in the type, or leftmost if the n_i are not ordered by $\multimap$. This means that the type next is always translated as $\forall\,(\alpha)$ $\forall\,(\beta)\ \alpha \to \beta$, not as $\forall\,(\beta)\ \forall\,(\alpha)\ \alpha \to\ \beta$.



Translating types Every node of χ can be translated to an $x\mathsf{ML}^{\mathsf{F}}$ type. Moreover, the translation is unique when using the ordering of the previous section. We follow the translation of $e\mathsf{ML}^{\mathsf{F}}$ types of (Yakobowski 2008), except for inert nodes which are inlined.

Each node n of χ is mapped to a type $\mathcal{S}(n)$ of $x\mathsf{ML}^{\mathsf{F}}$ as described in Figure 9. We assume that every node n in χ is in bijection with a type variable α_n. The translation uses the auxiliary functions $\mathcal{Q}(n)$ to build a sequence of type quantifications (one for each node flexibly bound to n), $\mathcal{R}(n)$ to inline n when it is rigid, and $\mathcal{T}(n)$ to build the bound of type variables in $\mathcal{Q}(n)$. The function $\mathcal{S}(n)$ distinguishes two special cases: when n is a G-node, it is translated by introducing the sequence of type quantifications $\mathcal{Q}(n)$ followed by the translation of $\langle n1\rangle$; when n is a regular node itself bound to a G-node, it is translated into α_n, which is always used in a context where α_n is bound. Otherwise, $\mathcal{S}(n)$ is $\mathcal{T}(n)$.

The notation $\bigwedge\!\!\!\!\diagdown(g)$used in Figure 8 can now be defined as $\Lambda(\mathcal{Q}(g))$. We also write $\mathcal{S}(\chi)$ for the translation $\mathcal{S}(\langle\rangle)$ of the root G-node of the whole constraint.

From instantiation witnesses to type instantiations The main part of the translation is the computation of the type instantiations from the instantiation witnesses. Let r be the root node of the expansion in χ^e. By construction, an instantiation witness Ω for e maps χ^e to χ. In fact, because Ω must leave χ unchanged, the sequence Ω may be decomposed into subsequences of the form:

(**1**) $\mathsf{Graft}(\sigma, n)$ or $\mathsf{Weaken}(n)$ with n in $\mathcal{I}(r)$;

(**2**) $\mathsf{Merge}(n, m)$ with n and m in $\mathcal{I}(r)$, and $m \prec n$;

(**3**) $\mathsf{Raise}(n)$ with $n \xrightarrow{+}\!\!\!\succ\!\!\longrightarrow r$;

(**4**) a sequence $(\mathsf{Raise}(n))^k$; $\mathsf{Merge}(n, m)$, with $n \in \mathcal{I}(r)$ and $m \notin \mathcal{I}(r)$. We write this sequence $\mathsf{RaiseMerge}(n, m)$ and see it as an atomic operation.

An operation $\mathsf{RaiseMerge}(n, m)$ lets n leaves the interior of r and be merged with some node m of χ bound above r. All other operations occur inside the interior of r. The grouping of operations in $\mathsf{RaiseMerge}(n, m)$ helps translating the subparts of instantiation witnesses that operate outside of $\mathcal{I}(r)$ into type instantiations.

Furthermore, since χ is a rigid presolution, we may also assume that (**5**) an operation $\mathsf{Weaken}(n)$ appears after all the other operations on a node below n. This ensures that Ω does not perform any operation under a rigidly bound node, which would not be expressible as an $x\mathsf{ML}^{\mathsf{F}}$ instantiation, as explained in §3.3.

We call *normalized* an instantiation witness that verifies the conditions (1)–(4) and (5) above. Normalized witnesses always exist. A constructive proof of this fact is given in (Yakobowski 2008).

Instantiation contexts In order to relate graphic nodes and $x\mathsf{ML}^{\mathsf{F}}$ bounds, we introduce one-hole *instantiation contexts* defined by the following grammar: $\mathcal{C} ::= \{\cdot\} \mid \forall\,(\geqslant \mathcal{C}) \mid \forall\,(\alpha \geqslant)\; \mathcal{C}$. We write $\mathcal{C}\{\phi\}$ for the replacement of the hole by the instantiation ϕ.

Consider a node n, and a node m flexibly transitively bound to n. Given our use of $\prec$ to order nodes, there exists a unique instantiation context $\mathcal{C}^n_m$ that can be used to descend in front of the quantification corresponding to m in $\mathcal{T}(n)$. For presolutions, and to avoid α-conversion related issues, we build instantiation contexts using variables whose names are based on the nodes they traverse.

For example, consider the constraint χ_p in Figure 8. The translation $\mathcal{T}(\langle\rangle)$ of the root G-node is $\forall\,(\alpha)\;\forall\,(\beta \geqslant \forall\,(\delta)\;\delta \to \alpha)\;\alpha \to \beta$. With the convention above, we have $\mathcal{C}^{\langle\rangle}_{\langle 11\rangle} = \{\cdot\}$, $\mathcal{C}^{\langle\rangle}_{\langle 12\rangle} = \forall\,(\alpha_{\langle 11\rangle} \geqslant)\;\{\cdot\}$ and $\mathcal{C}^{\langle\rangle}_{\langle 121\rangle} = \forall\,(\alpha_{\langle 11\rangle} \geqslant)\;\forall\,(\geqslant \{\cdot\})$.

Translating normalized derivations into instantiations Let us describe the translation of a normalized witness of $\chi^e \sqsubseteq \chi$ into an $x\mathsf{ML}^{\mathsf{F}}$ instantiation. We generalize the problem by translating a normalized witness Ω of $\xi \sqsubseteq \chi$

For a sequence of instructions:

$$\begin{array}{rcl} \Phi_\xi() & = & \mathbb{1} \\ \Phi_\xi(\omega; \Omega') & = & \Phi_\xi(\omega); \Phi_{\omega(\xi)}(\Omega') \end{array}$$

For an operation ω on a rigid node n:

$$\Phi_\xi(\omega) \;=\; \mathbb{1}$$

For an operation on the flexible root of the expansion r:

$$\begin{array}{rcl} \Phi_\xi(\mathsf{Graft}(\sigma, r)) & = & \mathcal{T}(\sigma) \\ \Phi_\xi(\mathsf{RaiseMerge}(r, m)) & = & !\alpha_m \\ \Phi_\xi(\mathsf{Weaken}(r)) & = & \mathbb{1} \end{array}$$

For an operation on a flexible node different from the root:

$$\begin{array}{rcl} \Phi_\xi(\mathsf{Graft}(\sigma, n)) & = & \mathcal{C}_n^r\{\forall\,(\geqslant \mathcal{T}(\sigma))\} \\ \Phi_\xi(\mathsf{RaiseMerge}(n, m)) & = & \mathcal{C}_n^r\{\langle !\alpha_m \rangle\} \\ \Phi_\xi(\mathsf{Merge}(n, m)) & = & \mathcal{C}_n^r\{\langle !\alpha_m \rangle\} \\ \Phi_\xi(\mathsf{Weaken}(n)) & = & \mathcal{C}_n^r\{\&\} \\ \Phi_\xi(\mathsf{Raise}(n)) & = & \mathcal{C}_m^r\{\otimes; \forall\,(\geqslant \mathcal{T}_\xi(n)); \\ & & \qquad \forall\,(\beta_n \geqslant)\, \mathcal{C}_n^m\{\langle !\beta_n \rangle\}\} \\ & & \text{where } m = \min_\prec\{m \mid n \rightarrowtail\rightarrowtail\leftarrowtail m \wedge n \prec m\} \end{array}$$

Figure 10: Translating normalized instance operations

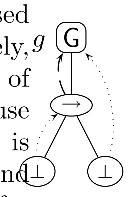
where ξ is such that $\chi^e \sqsubseteq \xi \sqsubseteq \chi$. Inside χ^e and ξ, we let r be the root of the expansion (inside χ, r is merged with d). The translation of $\xi \sqsubseteq \chi$ must witness the judgment $\Gamma_d \vdash \mathcal{T}_\xi(r) \leq \mathcal{T}_\chi(r)$ where Γ_d is the typing context for the node d. The translation of Ω, written $\Phi_\xi(\Omega)$, is defined by induction on Ω as described in Figure 10. The function Φ_ξ is overloaded to act on both an instance derivation and a single operation.

The translation of an instance derivation is defined recursively: the translation of an empty derivation is the identity instantiation $\mathbb{1}$; otherwise, Ω is of the form $(\omega; \Omega')$ and we return the composition of the translation of the operation ω followed by the translation of the instance derivation Ω' applied to the constraint $\omega(\xi)$.

The translation of an operation on a rigid node is the identity instantiation $\mathbb{1}$, as rigid bounds are inlined. Inert nodes have been weakened into rigid ones and locked nodes do not allow instance. Hence, the remaining and interesting part of the translation is a (single) operation applied to an instantiable node.

The translation of an instance operation on r (when r is flexible) is handled especially, as follows. The grafting of a type σ is translated to the instantiation τ—where τ is the translation of σ into $x\mathsf{ML}^\mathsf{F}$. A raise-merge of r with m is translated to $!\alpha_n$: it must be the last operation of the derivation Ω and α_m must be bound in the typing environment Γ_d; hence we may abstract the type of r under α_m. The weakening of r is translated to $\mathbb{1}$: it must be the next-to-the-last operation in the derivation Ω, before the merging of r with a rigidly bound node, and there is actually nothing to reflect in $x\mathsf{ML}^{\mathsf{F}}$, as the type of r itself is unchanged—only its binding flag in the expansion is.

In the remaining cases, the operation is applied to an instantiable node n. Since the derivation is normalized and n is not rigid, n must be transitively flexibly bound to r. Therefore, there exists an instantiation context $\mathcal{C}^r_n$ to reach the bound of α_n in $\mathcal{T}_\xi(r)$. The grafting of a type σ at n is translated to $\mathcal{C}^r_n\{\forall\,(\geqslant \mathcal{T}(\sigma))\}$ that transforms the bound $\bot$ of α_n into $\mathcal{T}(\sigma)$. The merging of n with a node m is translated to $\mathcal{C}^r_n\{\langle !\alpha_m\rangle\}$, which first abstracts the bound of α_n under the name α_m and immediately eliminates the quantification (we assume $m \prec n$). The translation is the same for a raise-merge, but α_m is bound in the typing environment instead of in $\mathcal{T}_\xi(r)$. The weakening of n is translated to $\mathcal{C}^r_n\{\forall\,(\geqslant \&)\}$. Finally, the translation of the raising of n is of the form $\mathcal{C}^r_m\{\invamp; \forall\,(\geqslant \mathcal{T}_\chi(n)); \phi\}$. We first insert a fresh quantification, bound by the type $\mathcal{T}_\xi(n)$, inside $\mathcal{T}_\xi(r)$. The difficulty consists in finding the node m in front of which to insert this quantification, so as to respect the ordering between bounds. Notice that the set $\{m \mid n \rightarrowtail\!\!\longrightarrow\rightarrowtail\!\!\longrightarrow\longleftarrow\!\!\prec m \wedge n \prec m\}$ contains at least the bound of n, hence its minimum m is well-defined. Then, the instantiation ϕ equal to $\forall\,(\beta_n \geqslant)\ \mathcal{C}^m_n\{\langle !\beta_n\rangle\}$ aliases the bound of n to the quantification just introduced and eliminates the resulting quantification. The net result of the whole type instantiation is that the type of n is introduced one level-higher than it previously was.



Reordering quantifiers It remains to define the notation $\Phi(e)$ used in Figure 7. We let Ω be a normalized witness for e. Unfortunately, we cannot simply use $\Phi_{\chi^e}(\Omega)$, as, in some cases, the type $\mathcal{T}_{\chi^e}(r)$ of g in the expansion does not correspond to $\mathcal{S}(g)$, regardless of our use of $\prec$. This can easily be seen in the example next, in which $\mathcal{S}(g)$ is $\forall\,(\beta)\ \forall\,(\alpha)\ \alpha \to \beta$: as we start by translating the flexible nodes bound on g, here $\langle g12\rangle$, before translating $\langle g1\rangle$; however, the expansion of g has type $\forall\,(\alpha)\ \forall\,(\beta)\ \alpha \to \beta$: the quantifiers appear in the opposite order. We believe that this difficulty is actually inherent to elaborating terms for languages with second-order polymorphism, in which second-order polymorphism can be kept local (as here for $\langle g11\rangle$), or be introduced by generalization (as for $\langle g12\rangle$). Thankfully, $\mathcal{T}_{\chi^e}(r)$ and $\mathcal{S}(g)$ may differ only by a reordering of quantifiers. In $x\mathsf{ML}^{\mathsf{F}}$, we can explicitly reorder them through the use of instantiations such as

$$\invamp; \forall\,(\geqslant \tau_\alpha);\ \ \invamp; \forall\,(\geqslant \tau_\beta);\ \ \forall\,(\beta \geqslant)\ \forall\,(\alpha \geqslant)\ \big(\langle !\alpha\rangle;\ \langle !\beta\rangle\big)$$

which commutes α and β in the type $\forall\,(\alpha \geqslant \tau_\alpha)\ \forall\,(\beta \geqslant \tau_\beta)\ \tau$. We write $\Sigma(g)$ the instantiation that transforms $\mathcal{S}(g)$ into $\mathcal{T}_{\chi^e}(r)$. Then, we define $\Phi(e)$ as $\Sigma(g); \Phi_{\chi^e}(\Omega)$.

Translating annotated terms As mentioned in §3.1, expressions such as $(b : \sigma)$ and $\lambda(y : \sigma)\ b$ are actually syntactic sugar, for $\kappa_\sigma\ b$ and $\lambda(y)\ \mathsf{let}\ y = \kappa_\sigma\ y\ \mathsf{in}\ b$, respectively. The translation $\mathcal{T}(\kappa_\sigma)$ of the type of the coercion function κ_σ in $x\mathsf{ML}^{\mathsf{F}}$ is $\forall\,(\alpha \geqslant \mathcal{T}(\sigma))\ \mathcal{T}(\sigma) \to \alpha$. Interestingly, coercion functions need not be primitive in $x\mathsf{ML^F}$—unlike in $e\mathsf{ML^F}$. Let id_κ be the expression $\Lambda(\alpha)\ \Lambda(\beta{\geqslant}\alpha)\ \lambda(x : \alpha)\ (x\,(!\beta))$. Then, define κ_σ as $\mathsf{id}_\kappa\langle \mathcal{T}(\sigma)\rangle$. Notice that κ_σ behaves as the identity function, as expected. Moreover, coercion functions can always be eliminated by reduction after the elaboration of the presolution, so that they have no runtime cost.

3.5 Soundness of the translation

Theorem 6 *Let b be an $e\mathsf{ML^F}$ term, χ a rigid presolution for b. The translation $[\![\, b \,]\!]$ of χ is well-typed in $x\mathsf{ML^F}$, of type $\mathcal{S}(\chi)$.*

Our translation preserves the type-erasure of programs (Lemma 7). Hence, the soundness of $x\mathsf{ML^F}$ also implies the soundness of $e\mathsf{ML^F}$—which had previously only been proved for the syntactic versions of $\mathsf{ML^F}$, but not for the most general, graphical version.

4 Discussion

4.1 Expressiveness of $x\mathsf{ML^F}$

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

For the sake of simplicity, we explain the difference between $x\mathsf{ML^F}$ and $i\mathsf{ML^F}$ the Curry-style version of $\mathsf{ML^F}$ (which has the same expressiveness as $e\mathsf{ML^F}$). Although syntactically identical, the types of $x\mathsf{ML^F}$ and of syntactic $i\mathsf{ML^F}$ 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 $i\mathsf{ML^F}$, 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 $x\mathsf{ML^F}$ and at least a superset of $\{\tau'' \to \tau' \mid \tau \leq \tau' \leq \tau''\}$. This level of generality cannot be expressed in $i\mathsf{ML^F}$. Interestingly, graphic types disallow alias bounds entirely, as they cannot even be expressed.

The current treatment of alias bounds in $x\mathsf{ML^F}$ is quite natural in a Church-style presentation. Surprisingly, it is also simpler than treating them as in $e\mathsf{ML^F}$. A restriction of $x\mathsf{ML^F}$ without alias bounds that is closed under reduction and in closer correspondence with $i\mathsf{ML^F}$ can still be defined a posteriori, by constraining the formation of terms, but the definition is contrived and unnatural.

Instead of restricting $x\mathsf{ML^F}$ to match the expressiveness of $i\mathsf{ML^F}$, a fair question is whether the treatment of alias bounds could be enhanced in $i\mathsf{ML^F}$—and

$e\mathsf{ML^F}$—to match the one in $x\mathsf{ML^F}$ without compromising type inference. This is worth further investigation.

4.2 Elaboration for other presentations of $\mathsf{ML^F}$

We have described the elaboration for the graphical, implicit version of $\mathsf{ML^F}$, since this is the most appropriate version for performing type inference. There are four presentations of $\mathsf{ML^F}$ depending on whether types are presented graphically or syntactically and whether annotations are explicit ($e\mathsf{ML^F}$) or implicit ($i\mathsf{ML^F}$). Our elaboration can be easily adapted to the three other presentations, with only minor differences, discussed below.

The graphical explicit version of $\mathsf{ML^F}$ is obtained by allowing the inverse of instance operations, but only on inert or rigid nodes. As a result of enlarging the instance relation, type inference becomes undecidable in $i\mathsf{ML^F}$. Still, the graphical framework of $e\mathsf{ML^F}$ applies to this variant, and a presolution in $e\mathsf{ML^F}$ is also a presolution in $i\mathsf{ML^F}$.

Interestingly, presolutions of $i\mathsf{ML^F}$ can also be elaborated into $x\mathsf{ML^F}$, as the difference between $e\mathsf{ML^F}$ and $i\mathsf{ML^F}$ lies in operations on inert and rigid nodes which are inlined in $x\mathsf{ML^F}$. The elaboration proceeds as for $e\mathsf{ML^F}$, by weakening presolutions into rigid ones, so that all inert nodes become rigid and will be inlined. The main difference lies in normalized derivations of instantiation edges (§3.4), which may contain new forms of operations in $i\mathsf{ML^F}$. However, those operations only occur on rigid nodes and are elaborated into identity type instantiations.

Translating syntactic versions of $\mathsf{ML^F}$ (whether implicit or explicit) into $x\mathsf{ML^F}$ might seem trivial at a cursory glance. However, this is not the case at all, and special care must also be taken because of the mismatch between type instance in $\mathsf{ML^F}$ and $x\mathsf{ML^F}$.

As for graphs, all rigid (in the case of $e\mathsf{ML^F}$) and inert types must be inlined, and types must be put in canonical form (based for instance on some total ordering of bound variables). This avoids the need for any form of equivalence or abstraction at places where it is not allowed in $x\mathsf{ML^F}$. Furthermore, alias bounds must also be inlined so as to preserve their intended semantics in $\mathsf{ML^F}$ (§4.1).

Once these precautions are carefully taken, the main part of the translation is however slightly simpler than in the graphical case, because instance derivations, which are the counterpart of instantiation witnesses, are closer to type instantiation in $x\mathsf{ML^F}$. In particular, the ordering of quantifiers has already been chosen in syntactic $e\mathsf{ML^F}$ derivations. However, this merely moves the task of consistently ordering quantifiers from the translation (in the graphical case) to type inference (in the syntactic case).

A similar translation should also be applicable to the language HML—an interesting variant of $\mathsf{ML^F}$ proposed by Leijen (2009) that is even more explicit than $e\mathsf{ML^F}$, but uses the simpler types of $i\mathsf{ML^F}$: at the price of adding extra annotations in source terms, HML needs not use rigid bounds at all. For the reasons developed above, we do not expect the elaboration for HML to be significantly simpler than for $i\mathsf{ML^F}$ or $e\mathsf{ML^F}$. However, its proofs of correctness might be simpler than the proof of correctness for $e\mathsf{ML^F}$ and itself simpler than the one for $i\mathsf{ML^F}$—since intuitively, the smaller the type equivalence, the simpler the proof of correctness. Alternatively, programs of HML could be elaborated indirectly by translating them into $e\mathsf{ML^F}$, then into $x\mathsf{ML^F}$.

4.3 Related works

Besides the several papers that describe variants of $\mathsf{ML^F}$, there are actually few related works.

Leijen and Löh (2005) have studied the extension of $\mathsf{ML^F}$ with qualified types, and as a subcase, the translation of $\mathsf{ML^F}$ 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 $\forall\,(\alpha)\ (\tau'_\star \rightarrow \alpha) \rightarrow \tau_\star$, 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 incidence 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.

Interestingly, while their translation and ours work on very different inputs—syntactic typing derivations in their case, graphic presolutions in ours—there are strong similarities between the two. The resemblance is even closer with the improved translation recently proposed by Leijen (2007), in which rigid bindings are inlined during the translation. As another example, we both canonically order quantifiers inside types. (However, our motivations are slightly different. We strive to reduce the number of quantifier reorderings, thus order all the quantifiers. Leijen uses only a weak canonical form, sufficient to obtain well-typed terms. This can result in some reorderings that are not present in our translation.)

4.4 Future works

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

While graphical type inference has been designed to keep maximal sharing of types during inference so as to have good practical complexity, our elaboration implementation reads back dags as trees and undo all the sharing carefully maintained during inference. Even with today's fast machines, this might be a problem when writing large, automatically generated programs. Hence, it would be worth maintaining the sharing during the translation, perhaps by adding type definitions to $x\mathsf{ML}^\mathsf{F}$.

It was somewhat of a surprise to realize that $x\mathsf{ML}^\mathsf{F}$ types are actually more expressive than $i\mathsf{ML}^\mathsf{F}$ ones, because of a different interpretation of alias bounds. While the interpretation of $x\mathsf{ML}^\mathsf{F}$ seems quite natural in an explicitly typed context, and is in fact similar to the interpretation of subtype bounds in $\mathsf{F}_{<:}$, the $e\mathsf{ML}^\mathsf{F}$ interpretation also seemed the obvious choice in the context of type inference. We have left for future work the question of whether the additional power brought by the $x\mathsf{ML}^\mathsf{F}$ could be returned back to $e\mathsf{ML}^\mathsf{F}$ while retaining type inference.

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) using term-level coercions (which should hopefully be semantics preserving) and $x\mathsf{ML}^\mathsf{F}$ which uses type-level instantiations (which are semantics preserving by construction). 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 abstract over the instantiation $!\alpha$ of type $\tau \to \alpha$.

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. Thus, 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 it we request all subterms to have identical types, it is not so clear if we only care about typability.

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.

Less ambitious is to directly generate smaller type instantiations when translating $e\mathsf{ML^F}$ presolutions into $x\mathsf{ML^F}$, by carefully selecting the instantiation witness to translate—as there usually exists more than one witness for a given instantiation edge. This amounts to using type derivations equivalence in $e\mathsf{ML^F}$ instead of observational equivalence in $x\mathsf{ML^F}$. Ideally, the latter should suffice. In practice, using just the former or the two combined might be simpler.

Extending $x\mathsf{ML^F}$ 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 $e\mathsf{ML^F}$ (Herms 2009).

Conclusion

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

We have described a translation of partially typed $e\mathsf{ML^F}$ programs into fully typed $x\mathsf{ML^F}$ ones. The translation preserves well-typedness and the type erasure of terms, which ensures the type soundness of $e\mathsf{ML^F}$. We have shown that $x\mathsf{ML^F}$ can be used as an internal language for $\mathsf{ML^F}$, with either call-by-value or call-by-name semantics, and also for the many restrictions of $\mathsf{ML^F}$ that have been proposed, including HML.

Hopefully, this will help the adoption of $\mathsf{ML^F}$ 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.

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

Mark P. Jones. A theory of qualified types. *Sci. Comput. Program.*, 22(3): 231–256, 1994.

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. 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 2008.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 1994.

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