A Church-Style Intermediate Language for MLF

Didier Rémy, Boris Yakobowski

A Church-Style Intermediate Language for MLF

Didier Rémy[a], Boris Yakobowski[b]

[a] *INRIA*
Rocquencourt - BP 105, 78153 Le Chesnay Cedex
[b] *CEA, LIST, Laboratoire Sûreté des Logiciels,*
Boîte 94, 91191 Gif-sur-Yvette Cedex, France.

Abstract

MLF is a type system that seamlessly merges ML-style implicit but second-class polymorphism with System-F explicit first-class polymorphism. We present xMLF, a Church-style version of MLF 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 xMLF is more general than type application in System F. We equip xMLF 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 MLF into xMLF, which shows that xMLF can be used as the internal language for MLF after type inference, and also ensures type soundness for the most expressive variant of MLF.

Keywords: MLF, System F, Types, Type Generalization, Type Instantiation, Retyping functions, Coercions, Type Soundness, Binders

Introduction

MLF (Le Botlan and Rémy, 2003, 2009; 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, 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 MLF 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 $\forall(\alpha \geqslant \forall(\beta)\ \beta \to \beta)\ \alpha \to \alpha$, which should be read "$\alpha \to \alpha$ *where* α *is any instance of* $\forall(\beta)\ \beta \to \beta$", can be assigned to choice id, and the two previous alternatives can be recovered *a posteriori* by choosing different instances for α.

Currently, the language MLF comes with a Curry-style version, iMLF, where no type information is needed and a type-inference version, eMLF, that requires partial type information (Le Botlan and Rémy, 2009). However, eMLF is not quite in Church style: a large amount of type information is still implicit, and partial type information cannot be easily maintained during reduction. Hence, while eMLF is a good surface language, it is not a good candidate for use as an internal language during the compilation process, where some program transformations, and perhaps some reduction steps, are being performed. This has been a problem for the adoption of MLF in the Haskell community (Peyton Jones, 2003), as the Haskell compilation chain uses an explicitly-typed internal 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 eMLF. 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 eMLF is equipped with reduction rules for type annotations. This has actu-

URL: `http://gallium.inria.fr/~remy` (Didier Rémy), `http://www.yakobowski.org` (Boris Yakobowski)

ally been considered in the original presentation of ML^F, but only with limited success. The reduction kept track of annotation sites during reduction; this showed, in particular, that no new annotation site needs to be introduced during reduction. Unfortunately, the exact form of annotations could not be maintained during reduction, by lack of an appropriate language to describe their computation. As a result, it has only been shown that some type derivation can be rebuilt after the reduction of a well-typed program, but without exhibiting an algorithm to compute it during reduction.

Independently, Rémy and Yakobowski (2008) have introduced graphic constraints, both to simplify the presentation of ML^F and to improve its type inference algorithm. This also resulted in a simpler and more expressive definition of ML^F. Hence, by $e\mathsf{ML}^\mathsf{F}$, we refer to the graphical presentation of ML^F rather then the original version. Consistently, $i\mathsf{ML}^\mathsf{F}$ refers to the graphic Curry's style version of $e\mathsf{ML}^\mathsf{F}$. We still use the generic name ML^F when the style of presentation does not matter.

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

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

The natural definition of $x\mathsf{ML}^\mathsf{F}$ is actually more expressive than that of ML^F. Still, we can restrict type-checking in $x\mathsf{ML}^\mathsf{F}$ so that well-typed terms are in closer correspondence with ML^F terms. This defines a well-behaved subset $x\mathsf{ML}^\mathsf{F}_\flat$ of $x\mathsf{ML}^\mathsf{F}$. Then, all three versions $i\mathsf{ML}^\mathsf{F}$, $e\mathsf{ML}^\mathsf{F}$ and $x\mathsf{ML}^\mathsf{F}_\flat$ have the same expressiveness, and only differ by the amount of type information: terms of $i\mathsf{ML}^\mathsf{F}$ contain none, terms of $e\mathsf{ML}^\mathsf{F}$contain some type annotations and no description of type instantiations, while $x\mathsf{ML}^\mathsf{F}$ contains all type annotations and a full description of all type instantiations.

A term of $x\mathsf{ML}^\mathsf{F}$ can easily be converted into an $e\mathsf{ML}^\mathsf{F}$ one by retaining type annotations, but dropping all other type information. The result may in turn be converted into a term of $i\mathsf{ML}^\mathsf{F}$ by further dropping all type annotations. Conversely, terms of $i\mathsf{ML}^\mathsf{F}$ cannot be automatically translated into terms of $e\mathsf{ML}^\mathsf{F}$, since type inference in $i\mathsf{ML}^\mathsf{F}$ is undecidable—some type annotations are required. Hence, source terms are terms of $e\mathsf{ML}^\mathsf{F}$: type inference can rebuild the type annotations that may be left implicit, or fail if mandatory type annotations have been omitted (or are incorrect). Terms of $e\mathsf{ML}^\mathsf{F}$—for which type inference succeeds—can then be elaborated into terms of $x\mathsf{ML}^\mathsf{F}$.

Outline. Perhaps surprisingly, the difficulty in defining an internal language for ML^F is not reflected in the internal language itself, which remains simple and easy to understand. Rather, the difficulties lie in the translation from $e\mathsf{ML}^\mathsf{F}$ to $x\mathsf{ML}^\mathsf{F}$, which is made somewhat complicated by many administrative details. Hence, we present $x\mathsf{ML}^\mathsf{F}$ first, and study its meta-theoretical properties independently of $e\mathsf{ML}^\mathsf{F}$. We then describe the elaboration of $e\mathsf{ML}^\mathsf{F}$ terms.

More precisely, the paper is organized as follows. We present $x\mathsf{ML}^\mathsf{F}$, its syntax and its static and dynamic semantics in §1. We study its main properties, including type soundness for different evaluations strategies in §2. The elaboration of $e\mathsf{ML}^\mathsf{F}$ programs into $x\mathsf{ML}^\mathsf{F}$ is described §3. We discuss the expressiveness of $x\mathsf{ML}^\mathsf{F}$ in §4 and related and future works in §5.

Proofs and implementation. The soundness proof of $x\mathsf{ML}^\mathsf{F}$ has been mechanized in the Coq proof assistant[1] and is briefly discussed in Appendix A. Other interesting proofs of §1 and §2 can be found in Appendix B, except for two results, which have already been proved by Manzonetto and Tranquilli (2010). Detailed proofs of §3 can all be found in the dissertation of Yakobowski (2008, Chapters 14 & 15), although for a slightly different—but equivalent—presentation. We do not reproduce them here, as they depend too much on the metatheoretical properties of $e\mathsf{ML}^\mathsf{F}$. The elaboration of $e\mathsf{ML}^\mathsf{F}$ into $x\mathsf{ML}^\mathsf{F}$ has been implemented in a prototype[2].

1. The calculus

1.1. Types, instantiations, terms, and typing environments

All the syntactic definitions of $x\mathsf{ML}^\mathsf{F}$ can be found in Figures 1 and 2. We assume given a countable collection of type variables written with letters α, β, γ, and δ. As usual, types include type variables and arrow types. Other type constructors will be added later—straightforwardly, as the arrow constructor receives no special treatment. Types also include a bottom type $\bot$ that corresponds to the System-F type $\forall\alpha.\alpha$. Finally, a type may also be a form of bounded quantification $\forall\,(\alpha \geqslant \tau)\ \tau'$, called *flexible* quantification, that generalizes the $\forall\alpha.\tau$ form of System F and, intuitively, restricts the variable α to range only over

[1] The Coq development is available at `http://www.yakobowski.org/publis/2010/xmlf-coq/`. Properties that have been mechanically verified in Coq are marked with the Coq symbol.

[2] Available at `http://gallium.inria.fr/~remy/mlf/proto/`.

$\alpha, \beta, \gamma, \delta$	:		**Type variable**
τ	::=		**Type**
	|	α	Type variable
	|	$\tau \to \tau$	Arrow type
	|	$\forall\,(\alpha \geqslant \tau)\ \tau$	Quantification
	|	$\bot$	Bottom type
ϕ	::=		**Instantiation**
	|	$@\tau$	Bottom
	|	$!\alpha$	Abstract
	|	$\forall\,(\geqslant \phi)$	Inside
	|	$\forall\,(\alpha \geqslant)\ \phi$	Under
	|	$\&$	$\forall$-elimination
	|	$⅋$	$\forall$-introduction
	|	$\phi;\phi$	Composition
	|	$\mathbb{1}$	Identity

Figure 1: Grammar of types and instantiations

a	::=		**Term**
	|	x	Variable
	|	$\lambda(x:\tau)\ a$	Function
	|	$a\ a$	Application
	|	$\Lambda(\alpha \geqslant \tau)\ a$	Type function
	|	$a\ \phi$	Instantiation
	|	$\mathsf{let}\ x = a\ \mathsf{in}\ a$	Let-binding
Γ	::=		**Environment**
	|	$\varnothing$	Empty
	|	$\Gamma, \alpha \geqslant \tau$	Type variable
	|	$\Gamma, x : \tau$	Term variable

Figure 2: Grammar of terms and typing contexts

$$\frac{}{\mathsf{wf}\,(\varnothing)} \qquad \frac{\alpha \notin \mathsf{dom}(\Gamma) \quad \mathsf{wf}\,(\Gamma) \quad \mathsf{ftv}(\tau) \subseteq \mathsf{dom}(\Gamma)}{\mathsf{wf}\,(\Gamma, \alpha \geqslant \tau)}$$

$$\frac{x \notin \mathsf{dom}(\Gamma) \quad \mathsf{wf}\,(\Gamma) \quad \mathsf{ftv}(\tau) \subseteq \mathsf{dom}(\Gamma)}{\mathsf{wf}\,(\Gamma, x : \tau)}$$

Figure 3: Well-formed environments

$$\textsc{Inst-Bot}\ \frac{}{\Gamma \vdash @\tau : \bot \leq \tau}$$

$$\textsc{Inst-Under}\ \frac{\Gamma, \alpha \geqslant \tau \vdash \phi : \tau_1 \leq \tau_2}{\Gamma \vdash \forall\,(\alpha \geqslant)\ \phi : \forall\,(\alpha \geqslant \tau)\ \tau_1 \leq \forall\,(\alpha \geqslant \tau)\ \tau_2}$$

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

$$\textsc{Inst-Intro}\ \frac{\alpha \notin \mathsf{ftv}(\tau)}{\Gamma \vdash ⅋ : \tau \leq \forall\,(\alpha \geqslant \bot)\ \tau} \qquad \textsc{Inst-Comp}\ \frac{\Gamma \vdash \phi_1 : \tau_1 \leq \tau_2 \quad \Gamma \vdash \phi_2 : \tau_2 \leq \tau_3}{\Gamma \vdash \phi_1;\phi_2 : \tau_1 \leq \tau_3}$$

$$\textsc{Inst-Elim}\ \frac{}{\Gamma \vdash \& : \forall\,(\alpha \geqslant \tau)\ \tau' \leq \tau'\{\alpha \leftarrow \tau\}} \qquad \textsc{Inst-Id}\ \frac{}{\Gamma \vdash \mathbb{1} : \tau \leq \tau}$$

Figure 4: Type instance

instances of τ. The variable α is bound in τ' but not in τ. We may just write $\forall\,(\alpha)\ \tau'$ when the bound τ is $\bot$.

In Church-style System F, type instantiation inside terms is simply type application $a\ \tau$. By contrast, in $x\mathsf{ML}^\mathsf{F}$, we use type instantiation $a\ \phi$ to detail 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 bindings, with two small differences: type instantiation $a\ \phi$ generalizes System-F type application as just described; and type abstractions are extended with an instance bound τ and written $\Lambda(\alpha \geqslant \tau)\ a$ where the type variable α is bound in a, but not in τ. We abbreviate $\Lambda(\alpha \geqslant \bot)\ a$ as $\Lambda(\alpha)\ a$, which simulates the type abstraction $\Lambda\alpha.\,a$ of System F.

We write $\mathsf{ftv}(\tau)$ and $\mathsf{ftv}(a)$ for the sets of type variables that appear free in τ and a, respectively. We identify types, instantiations, and terms up to the renaming of bound variables. The capture-avoiding substitution of an expression s_0 for a variable v inside an expression s is written $s\{v \leftarrow s_0\}$.

As usual, type environments assign types to program variables. However, instead of just listing type variables, as is the case in System F, they also assign them a type bound, using the form $\alpha \geqslant \tau$. We write $\mathsf{dom}(\Gamma)$ for the set of all term variables and type variables that are bound by Γ. We also assume that typing environments are *well-formed*, *i.e.* they do not bind twice the same variable and free type variables appearing in a type of the environment Γ are bound earlier in Γ. Well-formedness rules are given in Figure 3: the empty environment is well-formed; given a well-formed environment Γ, the relations $x \notin \mathsf{dom}(\Gamma)$, $\alpha \notin \mathsf{dom}(\Gamma)$, and $\mathsf{ftv}(\tau) \subseteq \mathsf{dom}(\Gamma)$ must hold to form environments $\Gamma, x : \tau$ and $\Gamma, \alpha \geqslant \tau$.

1.2. Instantiations

Instantiations ϕ are defined in Figure 1. Their typing, described in Figure C.19, 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'. (For conciseness, the syntax of instantiations uses mathematical symbols !, &, ⅋, *etc.* which have no connection at

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

Figure 5: Type instantiation (on types)

all with linear logic.)

The *bottom* instantiation $@\tau$ 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 and the environment in the premise of the rule Inst-Under is increased accordingly. The *quantifier introduction* $\wp$ 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 of the parametric minimum and comparison functions, and of the boolean conjunction:

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

Let ϕ be the instantiation $\forall\,(\geqslant @\mathsf{bool});\&$. Then, both $\vdash \phi : \tau_{\mathsf{min}} \leq \tau_{\mathsf{and}}$ 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 $\forall\,(\alpha \geqslant)\ (\forall\,(\geqslant @\alpha);\&)$ (the occurrence of α in the inside instantiation is bound by the under instantiation). Then, the relations $\vdash \phi' : \tau_K \leq \tau_{\mathsf{min}}$ holds.

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 previous instantiations ϕ and ϕ' can then be abbreviated as $\langle\mathsf{bool}\rangle$ and $\forall\,(\alpha \geqslant)\ \langle\alpha\rangle$.

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

$$
\textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{Let}\ \frac{\Gamma \vdash a : \tau \qquad \Gamma, x : \tau \vdash a' : \tau'}{\Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \tau'}
$$

$$
\textsc{App}\ \frac{\Gamma \vdash a_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_1\ a_2 : \tau_1}
$$

$$
\textsc{Abs}\ \frac{\Gamma, x : \tau \vdash a : \tau'}{\Gamma \vdash \lambda(x : \tau)\ a : \tau \to \tau'}
$$

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

Figure 6: Typing rules for $x\mathsf{ML}^\mathsf{F}$

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

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*[3] that, given an instantiation ϕ and a type τ, returns (if it exists) the unique type $\tau\ \phi$ such that $\Gamma \vdash \phi : \tau \leq \tau\ \phi$ holds for some Γ. An inductive definition of this function is given in Figure 5. Type instantiation is complete for type instance:

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

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, 2009)—they are richer, because type equivalence in $x\mathsf{ML}^\mathsf{F}$ is finer than type equivalence in $i\mathsf{ML}^\mathsf{F}$, as explained in §4.

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

Typing rules are defined in Figure 6. Compared with System F, the novelties are type abstraction and type in-

[3] There should never be any ambiguity between type instantiation $\tau\ \phi$ and instantiation of expressions $a\ \phi$; moreover, both operations have strong similarities and are closely related.

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

Lemma 3. *If $\Gamma \vdash a : \tau_1$ and $\Gamma \vdash a : \tau_2$, then $\tau_1 = \tau_2$. Coq*

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 can be easily synthesized by a typechecker.

Example. Let id stand for the identity $\Lambda(\alpha \geqslant \bot)\ \lambda(x : \alpha)\ x$ and τ_{id} be the type $\forall(\alpha \geqslant \bot)\ \alpha \to \alpha$. We have $\vdash \mathsf{id} : \tau_{\mathsf{id}}$—much as in System F, except that unconstrained universal variables are given the bound $\bot$. 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 $\forall(\beta \geqslant \bot)\ \beta \to \beta \to \beta$. This is again similar to its typing in System F. We abbreviate this type as τ_{choice}.

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 $\forall(\beta \geqslant \tau_{\mathsf{id}})\ \beta \to \beta$. Indeed, its typing derivation ends with:

$$\dfrac{\dfrac{\dfrac{\Gamma_\beta \vdash \mathsf{choice} : \tau_{\mathsf{choice}}}{\Gamma_\beta \vdash \langle\beta\rangle : \tau_{\mathsf{choice}} \leq \beta \to \beta \to \beta}\ \textsc{TApp} \qquad \dfrac{\Gamma_\beta \vdash \mathsf{id} : \tau_{\mathsf{id}}}{\Gamma_\beta \vdash\ !\beta : \tau_{\mathsf{id}} \leq \beta\ (\mathbf{1})}\ \textsc{TApp}}{\Gamma_\beta \vdash \mathsf{choice}\ \langle\beta\rangle : \beta \to \beta \to \beta}\ \textsc{TApp} \qquad \dfrac{\Gamma_\beta \vdash \mathsf{id}\ !\beta : \beta}{\Gamma_\beta \vdash \mathsf{id}\ !\beta : \beta}\ \textsc{App}}{\Gamma_\beta \vdash \mathsf{choice}\ \langle\beta\rangle\ (\mathsf{id}\ !\beta) : \beta \to \beta}\ \textsc{App}}{\Gamma \vdash \Lambda(\beta \geqslant \tau_{\mathsf{id}})\ \mathsf{choice}\ \langle\beta\rangle\ (\mathsf{id}\ !\beta) : \forall(\beta \geqslant \tau_{\mathsf{id}})\ \beta \to \beta}\ \textsc{Abs}$$

where Γ_β is $\Gamma, \beta \geqslant \tau_{\mathsf{id}}$ and the key judgment (1), which follows by Rule Inst-Abstr, says that type τ_{id} can be seen as type β whenever β is declared to be an instance of τ_{id}.

The term $\mathsf{choice\_id}$ may also be given weaker types by type instantiation. For example, $\mathsf{choice\_id}\ \&$ has type $(\forall(\alpha \geqslant \bot)\ \alpha \to \alpha) \to (\forall(\alpha \geqslant \bot)\ \alpha \to \alpha)$ as in System F, while $\mathsf{choice\_id}\ (\otimes; \forall(\gamma \geqslant)\ (\forall(\geqslant \langle\gamma\rangle); \&))$ has the ML type $\forall(\gamma \geqslant \bot)\ (\gamma \to \gamma) \to \gamma \to \gamma$. The former expression can be understood directly, by fixing β to its bound τ_{id}. The latter can be understood informally as the introduction of a free type variable γ and then the instantiation of the bound τ_{id} of β to the type $\gamma \to \gamma$, say τ_γ. Formally, the derivation is a little tedious. Let Γ be the typing environment $\gamma \geqslant \bot$.

$$\begin{array}{lll}
(\lambda(x : \tau)\ a_1)\ a_2 & \longrightarrow a_1\{x \leftarrow a_2\} & \\
\mathsf{let}\ x = a_2\ \mathsf{in}\ a_1 & \longrightarrow a_1\{x \leftarrow a_2\} & \\
a\ \mathbb{1} & \longrightarrow a & \\
a\ (\phi; \phi') & \longrightarrow a\ \phi\ (\phi') & \\
a\ \otimes & \longrightarrow \Lambda(\alpha \geqslant \bot)\ a & \alpha \notin \mathsf{ftv}(a) \\
(\Lambda(\alpha \geqslant \tau)\ a)\ \& & \longrightarrow a\{!\alpha \leftarrow \mathbb{1}\}\{\alpha \leftarrow \tau\} & \\
(\Lambda(\alpha \geqslant \tau)\ a)\ (\forall(\alpha \geqslant)\ \phi) & \longrightarrow \Lambda(\alpha \geqslant \tau)\ (a\ \phi) & \\
(\Lambda(\alpha \geqslant \tau)\ a)\ (\forall(\geqslant \phi)) & \longrightarrow \Lambda(\alpha \geqslant \tau\ \phi)\ a\{!\alpha \leftarrow \phi; !\alpha\} & \\
E[a] \longrightarrow E[a'] \quad \text{if} \quad a \longrightarrow a' &&
\end{array}$$

Figure 7: Reduction rules

First, we have

$$\begin{array}{lllll}
\Gamma \vdash & @\gamma & : & \bot & \leq \quad \gamma \quad (2 \\
\Gamma \vdash & \forall(\geqslant @\gamma) & : & \forall(\alpha \geqslant \bot)\ \alpha \to \alpha & \leq \quad \forall(\alpha \geqslant \gamma)\ \alpha \to \alpha (3 \\
\Gamma \vdash & \& & : & \forall(\alpha \geqslant \gamma)\ \alpha \to \alpha & \leq \quad \gamma \to \gamma \quad (4 \\
\Gamma \vdash & \forall(\geqslant @\gamma); \& & : & \forall(\alpha \geqslant \bot)\ \alpha \to \alpha & \leq \quad \gamma \to \gamma \quad (5
\end{array}$$

(2) is by rule Inst-Bot; (3) is by Inst-Inside and (2); (4) is by Inst-Elim; (5) is by Inst-Comp, (3), and (4). Then,

$$\begin{array}{lllll}
\Gamma \vdash & \langle\gamma\rangle & : & \tau_{\mathsf{id}} & \leq \quad \gamma \to \gamma \\
\Gamma \vdash & \forall(\geqslant \langle\gamma\rangle) & : & \forall(\beta \geqslant \tau_{\mathsf{id}})\ \beta \to \beta & \leq \quad \forall(\beta \geqslant \gamma \to \gamma) \\
\Gamma \vdash & \& & : & \forall(\beta \geqslant \gamma \to \gamma)\ \beta \to \beta & \leq \quad (\gamma \to \gamma) \to \gamma \\
\Gamma \vdash & \forall(\geqslant \langle\gamma\rangle); \& & : & \forall(\beta \geqslant \tau_{\mathsf{id}})\ \beta \to \beta & \leq \quad (\gamma \to \gamma) \to \gamma
\end{array}$$

(6) is an abbreviation of (5); (7) is by Inst-Inside; (8) is by Inst-Elim; (9) is by Inst-Comp, (7) and (8). By rule Inst-Under and (9), we have

$$\vdash \forall(\gamma \geqslant)\ (\forall(\geqslant \langle\gamma\rangle); \&) : \forall(\gamma \geqslant \bot)\ \forall(\beta \geqslant \tau_{\mathsf{id}})\ \beta \to \beta \ \leq\ \forall(\gamma \geqslant \bot)\ (\gamma \to \gamma) \to \gamma \to \quad (\mathbf{10})$$

Finally, by rule Inst-Intro, (10), and Inst-Comp, we have:

$$\vdash \otimes; \forall(\gamma \geqslant)\ (\forall(\geqslant \langle\gamma\rangle); \&) : \forall(\beta \geqslant \tau_{\mathsf{id}})\ \beta \to \beta \ \leq\ \forall(\gamma \geqslant \bot)\ (\gamma \to \gamma) \to \gamma$$

As illustrated on this rather simpler example, computing all intermediate steps of a type instantiation is very tedious for a human and usually harder than just checking type instantiation. However, $x\mathsf{ML}^\mathsf{F}$ is only meant to be used as an internal language by a machine.

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 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 7. 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 contain only CONTEXT and β-steps or ι-step, respectively. Finally, the reduction is the reflexive and transitive closure $\longrightarrow^\star$ of the one-step reduction relation.

Reduction of type instantiation. By definition, 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 for 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, on 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 instantiation $!\alpha$ is no more τ, but $\tau\ \phi$. Hence, in order to maintain well-typedness, all the occurrences of the instantiation $!\alpha$ in a must be simultaneously replaced by the instantiation $(\phi; !\alpha)$. Here, the instantiation $!\alpha$ is seen as an atomic construct, *i.e.* all occurrences of $!\alpha$ are substituted, while other occurrences of α (*i.e.* that are not part of $!\alpha$) are left unchanged. Formally, $a\{!\alpha_0 \leftarrow \phi_0\}$ is defined recursively, as described in Figure 8 (abbreviating $\{!\alpha_0 \leftarrow \phi_0\}$ by θ). The interesting lines are the two first ones of the second column, as other lines are just lifting the substitution from the leaves to types, type instantiations, and terms in the usual way.

As an example of ι-INSIDE, 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

$$\lambda(x : \tau \to \tau)\ \lambda(y : \bot)\ y\ (@(\tau \to \tau))\ (z\ \mathbb{1})$$

Finally, 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

Types

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

Terms

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

Type instantiations

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

Figure 8: Definition of $a\,\theta$, where θ is $\{!\alpha_0 \leftarrow \phi_0\}$

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 $\mathsf{choice\_id}$ (with $\forall\,(\alpha)\ \tau$ and $\Lambda(\alpha)\ a$ being abbreviations for $\forall\,(\alpha \geqslant \bot)\ \tau$ and $\Lambda(\alpha \geqslant \bot)\ a$). Here, all type instantiations are eliminated by reduction, but this is of course not always the case in general.

$$\begin{aligned} \mathsf{choice\_id}\ (\forall\,(\geqslant \langle\mathsf{int}\rangle); \&) &: (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int}) \\ &\longrightarrow^\star \lambda(y : \mathsf{int} \to \mathsf{int})\ \lambda(x : \mathsf{int}) \\ \mathsf{choice\_id}\ \& &: (\forall\,(\alpha)\ \alpha \to \alpha) \to (\forall\,(\alpha)\ \alpha \\ &\longrightarrow^\star \lambda(y : \forall\,(\alpha)\ \alpha \to \alpha)\ (\Lambda(\alpha) \\ \mathsf{choice\_id}\ (\parr; \forall\,(\gamma \geqslant)\ (\forall\,(\geqslant \langle\gamma\rangle); \&)) &: \forall\,(\gamma)\ (\gamma \to \gamma) \to (\gamma \to \gamma) \\ &\longrightarrow^\star \Lambda(\gamma)\ \lambda(y : \gamma \to \gamma)\ \lambda(x : \end{aligned}$$

1.5. System F as a subsystem of xML$^{\mathsf{F}}$

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 rules 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 one-step System-F β-reduction $(\Lambda(\alpha)\ a)\ \langle\tau\rangle \longrightarrow a\{\alpha \leftarrow \tau\}$ can be performed in $x\mathsf{ML}^{\mathsf{F}}$ in

three steps:

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

Equality (1) is by definition; step (2) is by ι-Seq; step (3) is by ι-Inside; step (5) is by ι-Elim; equalities (4) and (6) are by type instantiation and by the assumption that a is a term of System F thus in which $!\alpha$ cannot appear.

Conversely, if a term a is in System F, then its reduction steps in $x\mathsf{ML}^{\mathsf{F}}$ are all of these forms but possibly interleaved. Formally, the Church-Rosser property and the strong normalization lemma stated in §2.2 ensure that any reduction of a in $x\mathsf{ML}^{\mathsf{F}}$ will eventually terminate with the same normal form, hence with its normal form in System F.

2. Properties of reduction

2.1. Subject reduction

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

Lemma 4 (Weakening). *Let $\Gamma, \Gamma', \Gamma''$ be a well-formed environment.*
If $\Gamma, \Gamma'' \vdash \phi : \tau_1 \leq \tau_2$, then $\Gamma, \Gamma', \Gamma'' \vdash \phi : \tau_1 \leq \tau_2$.
If $\Gamma, \Gamma'' \vdash a : \tau'$, then $\Gamma, \Gamma', \Gamma'' \vdash a : \tau'$. Coq

Lemma 5 (Term substitution).
If $\Gamma, x : \tau', \Gamma' \vdash \phi : \tau_1 \leq \tau_2$ then $\Gamma, \Gamma' \vdash \phi : \tau_1 \leq \tau_2$.
Suppose $\Gamma \vdash a' : \tau'$; if $\Gamma, x : \tau', \Gamma' \vdash a : \tau$ then $\Gamma, \Gamma' \vdash a\{x \leftarrow a'\} : \tau$. Coq

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

Lemma 6 (Bound substitution).
Let ϑ 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 $\Gamma, \Gamma'\vartheta \vdash \phi\theta : \tau_1\vartheta \leq \tau_2\vartheta$.
If $\Gamma, \alpha \geqslant \tau, \Gamma' \vdash a : \tau'$ then $\Gamma, \Gamma'\vartheta \vdash a\theta : \tau'\vartheta$. Coq

The result below ensures in turn the correctness of rule ι-Inside.

Lemma 7 (Narrowing). *Assume $\Gamma \vdash \phi : \tau \leq \tau'$. Let θ be $\{!\alpha \leftarrow \phi; !\alpha\}$.*
If $\Gamma, \alpha \geqslant \tau, \Gamma' \vdash \phi' : \tau_1 \leq \tau_2$ then $\Gamma, \alpha \geqslant \tau', \Gamma' \vdash \phi'\theta : \tau_1 \leq \tau_2$.
If $\Gamma, \alpha \geqslant \tau, \Gamma' \vdash a : \tau''$ then $\Gamma, \alpha \geqslant \tau', \Gamma' \vdash a\theta : \tau''$. Coq

Subject reduction is an easy consequence of all these results.

Theorem 8 (Subject reduction).
If $\Gamma \vdash a : \tau$ and $a \longrightarrow a'$ then, $\Gamma \vdash a' : \tau$. Coq

2.2. Confluence

Theorem 9. *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). The proof is uninteresting and omitted here; it can be found in (Yakobowski, 2008).

Confluence means that β-reduction and ι-reduction are independent. For instance, ι-reductions can be performed under λ-abstractions as far as possible while keeping a weak evaluation strategy for β-reduction.

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

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

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. Termination of reduction

The termination of reduction has been proved by Manzonetto and Tranquilli (2010).

Theorem 10. *(Manzonetto-Tranquilli) The reduction $\longrightarrow$ is terminating.*

As a corollary of this result and of Theorem 9, we have immediately

Corollary 11. *The relation $\longrightarrow$ is strongly normalizing.*

The proof of Theorem 10 is by translation of $x\mathsf{ML}^{\mathsf{F}}$ into System F, where reductions are known to terminate, and by showing a simulation between reduction in $x\mathsf{ML}^{\mathsf{F}}$

and reduction of the elaborated term in System F. (This is also discussed in §5.1.) As a corollary, $\longrightarrow_\iota$ alone is also terminating. The termination of $\longrightarrow$ is useful but not critical, as $x\mathsf{ML}^\mathsf{F}$ is meant to be used in a language with general recursion. However, the termination of $\longrightarrow_\iota$ is essential for $x\mathsf{ML}^\mathsf{F}$ to have a type-erasure semantics.

2.4. Type-erasure semantics

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

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

It is immediate to verify that two terms related by ι-reduction have the same type erasure. Moreover, if a term 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.

Lemma 12. *If $a \longrightarrow_\iota a'$ then $\lceil a \rceil = \lceil a' \rceil$. If $a \longrightarrow_\beta a'$, then $\lceil a \rceil \longrightarrow_\beta \lceil a' \rceil$.*

The converse direction is also true:

Lemma 13. *(Manzonetto-Tranquilli) If $\lceil a \rceil \longrightarrow_\beta M$, then there exist a' and a'' such that $a \longrightarrow_\iota^* a' \longrightarrow_\beta a''$ and $\lceil a'' \rceil = M$.*

A proof has been given by Manzonetto and Tranquilli (2010, Appendix B[4]). Combining these two results ensures that $x\mathsf{ML}^\mathsf{F}$ has a type-erasure semantics.

2.5. 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 we restrict the semantics to a weak evaluation strategy. We then 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 $\kappa\ (\tau_1, \dots \tau_{|\kappa|})$. We write $\overline{\alpha}$ for a sequence of variables $\alpha_1, \dots \alpha_k$ and $\forall\,(\overline{\alpha})\ \tau$ for the type $\forall\,(\alpha_1) \dots \forall\,(\alpha_k)\ \tau$. 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 $\forall\,(\overline{\alpha})\ \tau_1 \to \dots \tau_{|c|} \to \tau_0$, where τ_0 is a constructed type (hence neither bottom, a variable or an arrow type) whenever the constant c is a constructor.

We distinguish a subset of terms, called *values* and written v, that are term abstractions, type abstractions, full or partial applications of constructors, or partial applications of primitives. We use an auxiliary letter w to characterize the arguments of functions, which differ for call-by-value and call-by-name strategies. In values, an application of a constant c can involve a series of type instantiations, but only evaluated ones and placed before all other arguments. Moreover, the application may only be partial whenever c is a primitive. Evaluated instantiations θ may be quantifier eliminations or either inside or under (general) instantiations. In particular, $a\ (@\tau)$ and $a\ (!\alpha)$ are *never* values. The grammar for values and evaluated instantiations is as follows:

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

Importantly, values cannot have type $\bot$:

Lemma 14. *If v is a value and if $\vdash v : \tau$, then τ is not $\bot$.*

(Proof p. 25)

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

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

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

Progress: Well-typed, full applications of primitives can be reduced, *i.e.*, for any term a of the form $f\ \ \theta_1\ \dots\ \ \theta_k\ w_1 \dots w_{|f|}$ 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 setting to a call-by-value semantics. In this case, arguments of applications in

[4] The indirect proof given in §4 is not correct, since it relies on the subject reduction property for their intermediate System Fc, which unfortunately does not hold.

values are themselves restricted to values, *i.e.* w is taken equal to v. Reduction rules of Figure 7 are modified as follows. 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-SEQ 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 $\longrightarrow_{\mathsf{v}}^{\star}$ the resulting reduction relation. It follows from the above restrictions that the reduction is deterministic. Moreover, since δ-reduction preserves typings, by assumption, the relation $\longrightarrow_{\mathsf{v}}^{\star}$ also preserves typings by Theorem 8. Hence, in combination with progress, stated next, the evaluation of well-typed terms "cannot go wrong".

Theorem 15 (Progress for call-by-value). *If $\Gamma_0 \vdash a : \tau$, then either a is a value or $a \longrightarrow_{\mathsf{v}}^{\star} a'$ for some a'.*

(Proof p. 25)

Call-by-value reduction and the value restriction

The value-restriction is the standard way of adding side effects in a call-by-value language. We 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, that cannot have direct or indirect side effects. Those contain at least value-forms, *i.e.* values and term variables, as well as their type-instantiations. In the case of $x\mathsf{ML}^{\mathsf{F}}$, which is a target language and not a source one, we obtain a restricted grammar of (potentially) expansive expressions a, and a subset which is constituted of non-expansive expressions u.

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

As usual, we restrict let-bound expressions to be non-expansive, since they implicitly contain a type generalization. Hence, a let-bound expression is expansive when its body is expansive—but it remains non-expansive when its body is non-expansive. 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 b, and similarly for applications of constants to expansive expressions.

Lemma 16, stated below, ensures two things: our restricted grammar has a meaning as a standalone language (as it is stable by reduction); and non-expansive expressions are closed by reduction and are thus harmless in presence of side-effects.

Lemma 16. *Expansive and non-expansive expressions are closed by call-by-value reduction.*

As an immediate consequence:

Corollary 17. *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

In call-by-name reduction semantics, values may contain applications of constants to arbitrary expressions—and not just to values. That is, we take a for w. The ι-reduction is restricted as for call-by-value, while $\longrightarrow_{\beta}$ is unchanged. However, evaluation contexts are now $E_n ::= [\cdot] \mid E_n\ a \mid E_n\ \phi$.

We write $\longrightarrow_{\mathsf{n}}^{\star}$ the resulting reduction relation. As for call-by-value, it is deterministic by construction and preserves typings. Moreover, it may always progress. Hence, call-by-name evaluation of well-typed terms "cannot go wrong".

Theorem 18 (Progress for call-by-name). *If $\Gamma_0 \vdash a : \tau$, then either a is a value or $a \longrightarrow_{\mathsf{n}}^{\star} a'$ for some a'.*

(Proof p. 25)

3. Elaboration of graphical $e\mathsf{ML}^\mathsf{F}$ into $x\mathsf{ML}^\mathsf{F}$

To verify that, as expected, $x\mathsf{ML}^\mathsf{F}$ can be used as an internal language for $e\mathsf{ML}^\mathsf{F}$, we now exhibit a type-preserving type-erasure-preserving translation from $e\mathsf{ML}^\mathsf{F}$ to $x\mathsf{ML}^\mathsf{F}$. We use the graphic constraint presentation of $e\mathsf{ML}^\mathsf{F}$ (Rémy and Yakobowski, 2008; Yakobowski, 2008) which is more general than the syntactic presentation (Le Botlan and Rémy, 2003, 2009) and also better suited for type inference.

The elaboration of $e\mathsf{ML}^\mathsf{F}$ into $x\mathsf{ML}^\mathsf{F}$ proceeds in two phases. The first phrase is just type inference in $e\mathsf{ML}^\mathsf{F}$, described by Rémy and Yakobowski (2008) and Yakobowski (2008). A source program of $e\mathsf{ML}^\mathsf{F}$ is translated into a typing constraint, which can be seen as a decoration of the source program with (1) placeholders for missing types, and (2) type instantiation constraints that relate types (either known or unknown). The constraint is then solved, filling in all unknown types so that all type instantiation constraints become valid. The result of type inference is called a presolution.

The second phase translates a presolution into a term of $x\mathsf{ML}^\mathsf{F}$. The main difficulty is to infer for each instantiation constraint a precise description of the type instantiation steps. Interestingly, this is done by replaying type inference with an instrumented algorithm. More precisely, the instantiation steps are extracted from the proof that the presolution found by type inference is indeed in solved form. It then remains to translate the instrumented presolution, which is represented graphically, into a syntactic form, *i.e.* a term of $x\mathsf{ML}^\mathsf{F}$. This second phase is a form of compilation, which is technically not very deep, but meticulous.

Since the elaboration is based on—and starts with—type inference, it contains many details that require some minimal understanding of $e\mathsf{ML}^\mathsf{F}$. Hence we present an overview of $e\mathsf{ML}^\mathsf{F}$. Sill, other reading might help (Rémy and Yakobowski, 2007, 2008; Yakobowski, 2008). As no other part depends on §3, most details (or even the whole section) can also be skipped in a first reading of the paper.

Outline. We first review the graphic constraints type inference framework (§3.1); we then present the main steps of the translation (§3.2); finally, we describe the key steps in details (§3.3-3.5). The elaboration has been implemented in a prototype by Scherer (2010a).

3.1. An overview of graphical $e\mathsf{ML}^\mathsf{F}$

A full presentation of graphical $e\mathsf{ML}^\mathsf{F}$ is out of the scope of this paper. In this section, we only remind the key points about graphic types and associated type instance, which is the basis of the elaboration algorithm. We put more emphasis on the aspects of graphic types that either depart significantly from more traditional syntactic presentation of types, or that play a key role in understanding the elaboration process. Detailed presentations can be found in (Rémy and Yakobowski, 2007, 2008; Yakobowski, 2008).

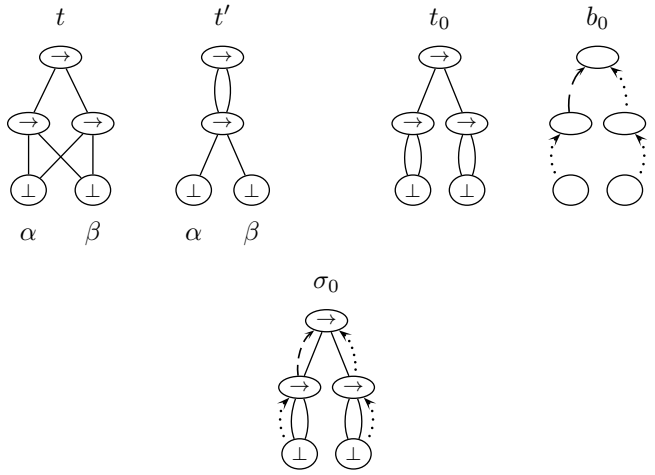


Figure 9: Dags and graphic types

3.1.1. Graphic types

Types of graphical $e\mathsf{ML}^\mathsf{F}$ 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 polymorphism.

Term-dags are just dag representations of usual tree-like types where all occurrences of the same variable are 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 $n \overset{i}{\circ\!\!\longrightarrow} m$, 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 $\langle\rangle$. On pictures, edges are drawn with plain lines, oriented downwards; we leave i implicit, as outgoing edges are always drawn from left to right.

Example 1. *The dag t on Figure 9 represents the first-order type $(\alpha \to \beta) \to (\alpha \to \beta)$. The nodes $\langle 11 \rangle$ and $\langle 22 \rangle$ are variables (the names α and β are here to help reading the figure, but formally they are not part of the graphic type). Compared with the tree notation, leaves representing the same variable are merged together; the names of leaves are left anonymous. That is, paths* 11 *and* 21 *lead to the same node, which can therefore be designated by $\langle 11 \rangle$ or $\langle 21 \rangle$, indifferently. Similarly, paths* 12 *and* 22 *lead to the same node.*

The dag structure also allows sharing internal nodes whose subtrees are identical as described by the dag t' where nodes $\langle 1 \rangle$ and $\langle 2 \rangle$ coincide. The dag t' could be syntactically written as $(\mathsf{let}\ \gamma = \alpha \to \beta\ \mathsf{in}\ \gamma \to \gamma)$. *In fact, sharing of internal nodes is a key to the efficient implementation of unification algorithms on first-order types. Those typically see t' as an* instance *of t, but not the converse; thus sharing can only be increased, and never lost. However, this refinement of the instance relation needs not be revealed externally, and dag t' can be displayed as dag t by splitting (or just reading back) shared internal nodes into separate*

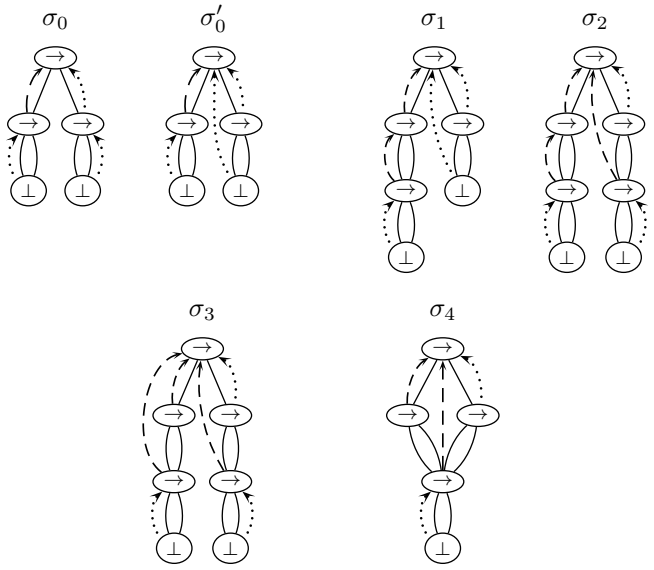


Figure 10: Examples of instance on graphic types

ones.

The second component of graphic types, 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 $\geqslant$ or $=$, respectively. On drawings, these flags are represented by dotted or dashed lines, respectively. We use the flag metavariable $\diamond$ to range over $\geqslant$ and $=$.

Example 2. *Consider the graphic type σ_0 of Figure 9. It is the superposition of the first-order term-dag t_0 and the binding tree b_0. The edge $\langle 22\rangle \overset{\geqslant}{\rightarrowtail} \langle 2\rangle$ is a flexible binding edge (the rightmost lowermost one), while $\langle 1\rangle \overset{=}{\rightarrowtail} \langle\rangle$ is a rigid binding edge (the leftmost uppermost one) and $\langle 1\rangle \overset{2}{\circ\!\!\rightarrow} \langle 12\rangle$ is a structure edge.*

Binding edges express polymorphism. They are oriented, and the target of the edge indicates the place where the binding occurs. The node at the source of the edge represents the variable being introduced, while the subtree at that node is the bound of that variable. Binding edges are of two kinds: a *rigid* edge means that polymorphism is required; typically, it is used for 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.

Example 3 (cont.). *The type σ_0 of Figures 9 and 10 describes a function f 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, 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, which is not as polymorphic as required.*

The type σ'_0 of Figure 10 describes a polymorphic function that, given a type γ, expects an argument of type $\forall(\alpha)\ \alpha \to \alpha$ and returns a value of type $\gamma \to \gamma$. In particular, σ'_0 is strictly less polymorphic than σ_0, as in System-F, since $\gamma \to \gamma$ is a strict instance of $\forall(\beta)\ \beta \to \beta$.

Rigid bounds arise from type annotations: the principal type of a term that contains no type annotations (in an environment that contains no types with rigid bounds), uses only flexible bounds. That is, required polymorphism may be offered by type inference, but never requested automatically.

Classifying nodes. For the purpose of defining type instance, we distinguish four kinds of nodes according to their position in the binding tree. The kind of each node is used below to determine how they can be transformed during type instantiations. Hence, this classification plays an important role in the translation.

Nodes on which no variable is transitively flexibly bound are called *inert*, as they neither hold nor control polymorphism. They will be discussed in detail further on. All other nodes hold or control some 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 in the next section; in $x\mathsf{ML}^\mathsf{F}$ these nodes correspond to parts of types that can be transformed by a suitable instantiation expression. Nodes whose binding edge is rigid are called *restricted*, because they cannot be grafted; in $x\mathsf{ML}^\mathsf{F}$ they roughly 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 roughly correspond to polymorphic types occurring in the bound of quantifiers themselves under some arrow type—they offer polymorphism that is requested and cannot be diminished.

Example 4 (cont.). *In the type σ'_0 of Figure 10, the node $\langle 2\rangle$ is inert, $\langle 21\rangle$ is instantiable, $\langle 1\rangle$ is restricted and $\langle 11\rangle$ is locked.*

Type instance. The *instance relation* on graphic types, written $\sqsubseteq$, can be described as the composition of four atomic operations: *grafting*, *merging*, *raising*, and *weakening*. All four operations are detailed below, and depicted schematically in Figure 11. In the figure, we use the following conventions to constrain the position of nodes in the binding tree: the green (or light gray) node with dotted border is instantiable; blue (or darker gray) nodes with double-line borders are anything but locked; small white nodes are unconstrained.

- $\mathsf{Graft}(\sigma, n)$, called *grafting*, replaces an instantiable bottom node n by a closed graph σ. Grafting corresponds

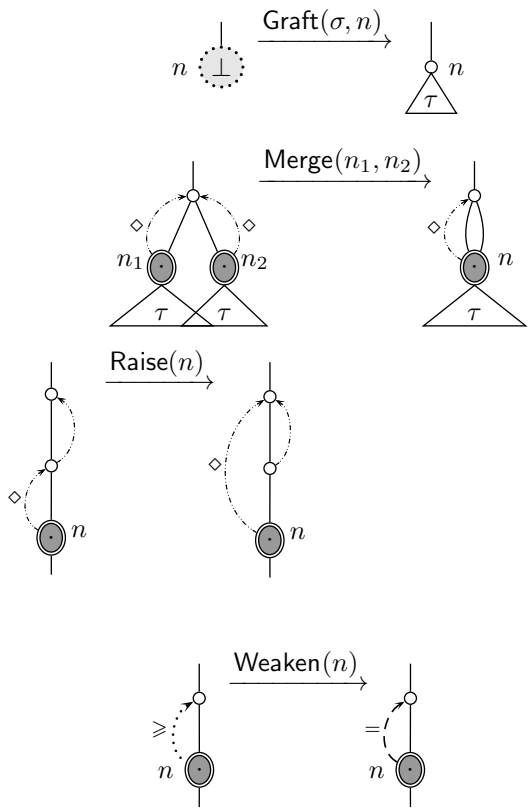


Figure 11: Atomic graphic instance operations

to the Inst-Bot rule of $x\mathsf{ML}^\mathsf{F}$. That is, $\Gamma \vdash @\tau : \bot \leq \tau$ where τ is the type describing the graph σ.

• $\mathsf{Merge}(n_1, n_2)$, called *merging*, fuses two nodes n_1 and n_2 that are not locked and have identical subgraphs. After merging, the subgraphs will thus be shared and can only be instantiated synchronously. In $x\mathsf{ML}^\mathsf{F}$ terms, it replaces two identical quantifications by an unique one, as in

$$\Gamma \vdash \phi : \forall\,(\alpha \geqslant \tau)\ \forall\,(\beta \geqslant \tau)\ \tau' \leq \forall\,(\alpha \geqslant \tau)\ \tau'\{\beta \leftarrow \alpha\}$$

with, for instance, ϕ equal to $\forall\,(\alpha \geqslant)\ (\forall\,(\geqslant !\alpha); \&)$.

• $\mathsf{Raise}(n)$, called *raising*, makes the binding of a node n that is not locked slide other the binding edge above it. Raising corresponds to a scope extrusion in $x\mathsf{ML}^\mathsf{F}$, as in

$$\Gamma \vdash \phi : \forall\,(\alpha \geqslant \forall\,(\beta \geqslant \tau)\ \tau')\ \tau'' \leq \forall\,(\beta \geqslant \tau)\ \forall\,(\alpha \geqslant \tau')\ \tau''$$

with, for instance, ϕ equal to $\Re; \forall\,(\geqslant @\tau); \forall\,(\beta \geqslant)\ (\forall\,(\geqslant \forall\,(\geqslant !\beta); \&))$.

• $\mathsf{Weaken}(n)$, called *weakening*, changes the binding of a flexible node n that is not locked into a rigid one. This freezes the subgraph under the node, preventing further instance operations on non-inert nodes, and all graftings. When this operation occurs on an instantiable node, it corresponds to the $x\mathsf{ML}^\mathsf{F}$ Inst-Elim instantiation:

$$\Gamma \vdash \& : \forall\,(\alpha \geqslant \tau)\ \tau' \leq \tau'\{\alpha \leftarrow \tau\}$$

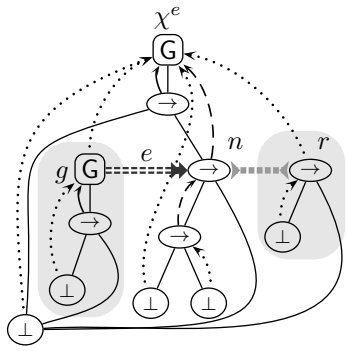


Figure 12: Constraints and expansion

Notice that grafting and merging do not change the bindings of existing nodes, while conversely, raising and weakening only change the bindings of existing nodes.

Example 5 (cont.). *The type σ'_0 of Figure 10 is an instance of σ_0 obtained by raising $\langle 21\rangle$. The type σ_4 is an instance of σ_1, obtained by grafting then weakening $\langle 21\rangle$ (resulting in σ_2), raising the node $\langle 11\rangle$ (which gives σ_3), and finally merging $\langle 11\rangle$ and $\langle 21\rangle$. Letting σ be the graph corresponding to $\forall\,(\alpha)\ \alpha \to \alpha$, we may formally write:*

$$\sigma_1 \xrightarrow{\mathsf{Graft}(\sigma, \langle 21\rangle)} \xrightarrow{\mathsf{Weaken}(\langle 21\rangle)} \sigma_2 \xrightarrow{\mathsf{Raise}(\langle 11\rangle)}$$

$$\sigma_3 \xrightarrow{\mathsf{Merge}(\langle 11\rangle, \langle 21\rangle)} \sigma_4$$

Hence, the instance $g_1 \sqsubseteq g_2$ is witnessed by the transformation

$$\mathsf{Graft}(\sigma, \langle 21\rangle); \mathsf{Weaken}(\langle 21\rangle); \mathsf{Raise}(\langle 11\rangle); \mathsf{Merge}(\langle 11\rangle, \langle 21\rangle)$$

where ";" is the composition with arguments given in reverse order.

On the importance of inert nodes. While inert nodes carry no polymorphism, it is important to treat them especially—so as to allow slightly more instance operations. Intuitively, since these nodes carry no polymorphism, they need not be shared, nor do they need a binding edge. However, it is technically more regular to let every node but the root node be bound to some other node, which we do. Furthermore, we only allow raising, merging or weakening those nodes, not the converse operations; §3.3 will justify why this is technically possible.

3.1.2. Type constraints

Type constraints are used to formalize ML^F typing problems. They generalize graphic types by adding new forms of edges, called constraint edges. These can be either *unification edges* ⟩┈┈⟨ or *instantiation edges* ┈┈▸▸. They also generalize let-constraints that have been proposed for type inference in ML by Pottier and Rémy (2005). 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 Figure 12. The instance on type constraints is exactly as on graphic types—constraint edges are just preserved.

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 $g \dashrightarrow n$ 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 (defined below) is an instance of χ.

A type constraint is solved when all of its constraint edges are solved. A *presolution* of a constraint is one of its solved instances. 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. In a constraint χ, consider an instantiation edge e defined as $g \dashrightarrow n$. 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 r 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[5] (but are instead shared with the original).

Example 6. *Let us consider the expansion χ^e of Figure 12. 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 leftmost highlighted nodes. 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 r is the copy of $\langle g1\rangle$. It is bound to the bound of n and connected to n by an unification edge.*

By definition, we say that an instantiation edge e is *solved* when χ is an instance of χ^e. This indeed means that the subtype constrained by the instantiation edge is less general than the type scheme at the origin of the edge—as a copy of this scheme can be instantiated into the subtype. We call *instantiation witness* an instance derivation of $\chi^e \sqsubseteq \chi$ for a solved instantiation edge e.

[5]Readers familiar with MLF (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, for conciseness.

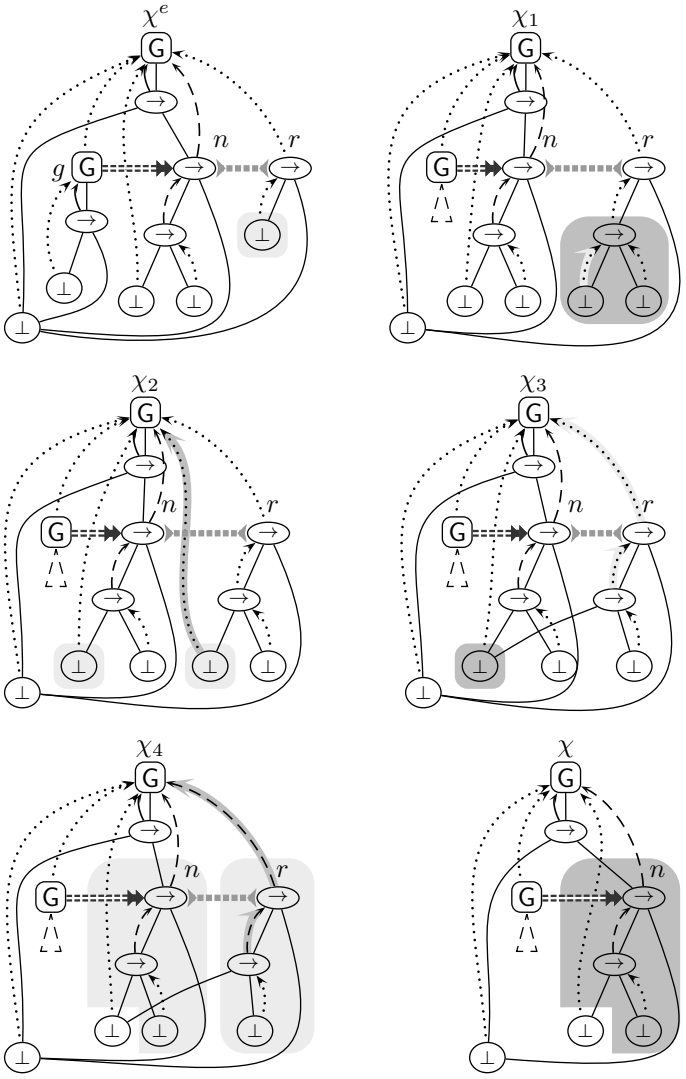


Figure 13: Example of solved instantiation edge

Example 7 (cont.). *In Figure 12, χ is an instance of χ^e—hence, the edge e is solved. This is witnessed by the sequence of transformations given below and depicted in Figure 13.*

All nodes below g are invariant during the transformations and are elided (represented as the ⚠ subtree) in all other constraints, for conciseness. Nodes or edges about to change are highlighted in green or in light gray, while those that have just changed are highlighted in red or in dark gray.

Grafting $\forall(\alpha)\,\forall(\beta)\,\alpha \to \beta$ under $\langle r1\rangle$ in χ^e leads to χ_1; raising $\langle r11\rangle$ twice gives χ_2; mergings nodes $\langle r11\rangle$ and $\langle n11\rangle$ gives χ_3; weakening node $\langle r1\rangle$, then node $\langle r\rangle$ leads to χ_4; finally, by merging n and r, which is possible as the two subgraphs under them are equal, we end up with exactly χ.

Formally, this is the transformation Ω:

$$\mathsf{Graft}(\forall(\alpha)\,\forall(\beta)\,\alpha \to \beta, \langle r1\rangle)\ ;\ \mathsf{Raise}(\langle r11\rangle)\ ;\ \mathsf{Raise}(\langle r11\rangle)\ ;$$
$$\mathsf{Merge}(\langle r11\rangle, \langle n11\rangle)\ ;\ \mathsf{Weaken}(\langle r1\rangle)\ ;\ \mathsf{Weaken}(r)\ ;\ \mathsf{Merge}(r, n)$$

3.1.3. From λ-terms to typing constraints

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

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

Type inference is performed by translating a source term into a type constraint, solving the constraint into a (principal) presolution, from which a (principal) solution can easily be read.

Type constraints are generated 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 are mapped to different nodes. (Formally, occurrences may be identified by their path to the root of the type constraint.) We let y and z stand for λ-bound and let-bound variables, respectively. We assume that the source term has been renamed so that every bound variable is distinct from all others.

Constraint generation is described on the top of Figure 14: each case refers to the expression on the left-hand side of the corresponding equality[6] at the bottom of the Figure. The unification edge u_y in (1) links the node that encodes an occurrence of a λ-bound variable y to the node y generated in (4) by the translation of the abstraction binding y. The instantiation edge e_z ending in (2) is coming from the G-node labeled b_1 generated in (3) by the translation of the let expression binding z. The type of an abstraction $\lambda(y)\ b$ is an arrow type whose domain is the type of y and codomain is an instance of the type of b, as witnessed by the edge e (4). The type for an application $b_1\ b_2$ is the codomain of an instance of the type of b_1, which must itself be an arrow type whose domain is an instance of the type of b_2 (5). The type of a let-expression $\mathsf{let}\ x = b_1\ \mathsf{in}\ b_2$ is just an instance of the type of b_2: as explained above, the constraints b_2 will contain, for every occurrence of x in b_2, one instance edge coming from the type of b_1 and ending at that occurrence. The typing constraint for let-expressions could be optimized to avoid taking an additional instance of b_2, as done in (Rémy and Yakobowski, 2008; Yakobowski, 2008). The advantage of this unoptimized version, which still preserves the complexity of type inference, is that every subexpression introduces exactly one G-node; this establishes a one-to-one mapping between subexpressions and G-nodes that is preserved during constraint resolution (G-nodes are never merged) and helps define the elaboration after constraint resolution.

Example 8. *The typing constraint χ for the term $\lambda(x)\ \lambda(y)\ x$ is described on the left-hand side of Figure 15. One of its presolutions χ_p is drawn on the middle. (We have dropped the mapping of expressions to G-nodes for conciseness, and labeled some binding edges that will appear in the $x\mathsf{ML}^\mathsf{F}$ translation.) This is not the most general presolution, as some 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 $x\mathsf{ML}^\mathsf{F}$.*

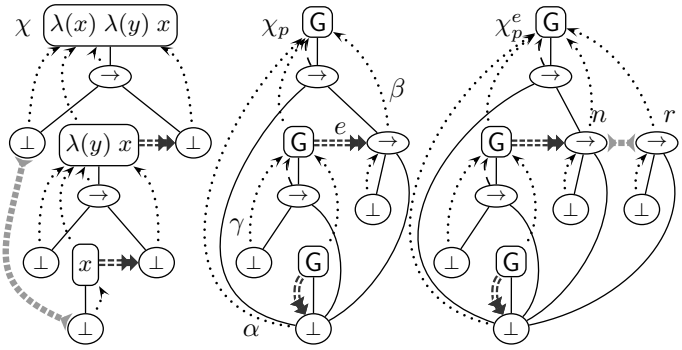


Figure 15: Typing constraints for $\lambda(x)\ \lambda(y)\ x$.

While type inference is out of the scope of this work, we may however easily check *that χ_p is 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^e_p drawn on the right-hand side, that is, exhibit a sequence of atomic instance operations that transforms χ^e_p into χ_p. Here, the obvious solution is just to merge the two nodes related by the unification edge,* i.e. $\mathsf{Merge}(n, r)$.

Annotated expressions. 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[7], where κ_σ is a coercion function that has type $\forall(\alpha \geqslant \sigma)\ \sigma \to \alpha$ in $x\mathsf{ML}^\mathsf{F}$; those coercion functions are discussed in more detail in §3.6.

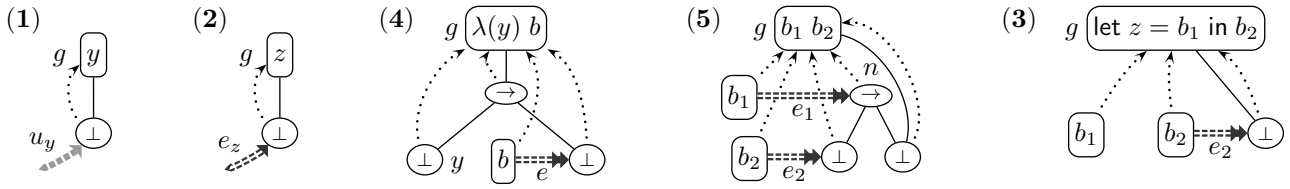
Both constructs are desugared before the translation into constraints. The effect of rebinding x to $\kappa_\sigma\ x$ is to request the parameter x to be of type σ and simultaneously let all occurrences of x in b be typed with possibly different instances of σ. By contrast, $\lambda(x)\ b$, without an annotation, forces the parameter x and all occurrences of x in b to have exactly the same type.

3.2. An overview of the translation to $x\mathsf{ML}^\mathsf{F}$

The elaboration of an $e\mathsf{ML}^\mathsf{F}$ term b to $x\mathsf{ML}^\mathsf{F}$ is based on a presolution χ of the typing constraint for b. The translation is based on presolutions rather than solutions, since presolutions still contain the original subconstraints (unlike solutions, which only retain the final type). While typing constraints have principal presolutions, any presolution—not merely the principal one that is returned

[6] The right-hand side is the elaboration of the left-hand side, which will be explained in the next section.

[7] The expression $\lambda(x)\ \mathsf{let}\ x = \kappa_\sigma\ x\ \mathsf{in}\ b$ is equal to $\lambda(y)\ \mathsf{let}\ x = \kappa_\sigma\ y\ \mathsf{in}\ b$ whenever y does not appear free in b; using the same variable x for y avoids the side condition and so makes the syntactic sugar a purely local transformation.



$$
\begin{array}{rcl}
[\![y]\!] & =^{(1)} & y \\
[\![z]\!] & =^{(2)} & \bigwedge\!\!\backslash(g)\ z\ (\Phi(e_z)) \\
[\![\lambda(y)\ b]\!] & =^{(4)} & \bigwedge\!\!\backslash(g)\ \lambda(y : \mathcal{T}(y))\ [\![b]\!]\ (\Phi(e)) \\
[\![b_1\ b_2]\!] & =^{(5)} & \bigwedge\!\!\backslash(g)\ [\![b_1]\!]\ (\Phi(e_1))\ \ ([\![b_2]\!]\ (\Phi(e_2))) \\
[\![\mathsf{let}\ z = b_1\ \mathsf{in}\ b_2]\!] & =^{(3)} & \bigwedge\!\!\backslash(g)\ \mathsf{let}\ z = [\![b_1]\!]\ \mathsf{in}\ [\![b_2]\!]\ (\Phi(e_2))
\end{array}
$$

Figure 14: Constraint generation and translation of presolutions

by type inference—can be translated. However, presolutions must be slightly transformed into *rigid* presolutions before translating them, as explained in §3.3—but we may ignore this minor detail for the moment.

Given an original program b and a (rigid) presolution of the graphic constraint for b, the translation is inductively defined on the structure of b, reading auxiliary information on the corresponding nodes in the presolution; we build this way the type of function parameters, type abstractions, and type instantiations. 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 the top of Figure 14 when defining the translation (hence both top and bottom parts of Figure 14 should now be read in parallel to understand the translation). There are two key ingredients:

- For each instantiation edge e of the form $g \dashrightarrow n$, 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 and §3.5.

- For each flexible binding edge to a G-node $n \rightarrowtail g$, 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 $\bigwedge\!\!\backslash(g)$ to refer to the sequence of all such quantifications at the level of g, which is a binding prefix of the form $\Lambda(\alpha_1 \geqslant \tau_1)\ \ldots \Lambda(\alpha_q \geqslant \tau_q)$ that will be precisely defined in §3.4.

  (Conversely, rigid bindings, which are only useful to make type inference decidable, are inlined during the translation and thus do not give rise to type quantifications.)

The translation is given in Figure 14. We let $\bigwedge\!\!\backslash(g)$ and $\Phi(e)$ abstract for the moment. They will be defined in sections 3.4 and 3.5, respectively.

The translation of a λ-bound variable y (1) is itself. Indeed, the G-node y is always monomorphic and there is no polymorphism to introduce; moreover, as the type of y in the presolution is its only instance, there is no need to add a type instantiation. For all other cases, the translation is of the form $\bigwedge\!\!\backslash(g)\ b'$, g being the G-node for b. Indeed, generalization is needed in ML^F for let-bound expressions (as in ML) and also for applications and abstractions (unlike in ML).

An occurrence of a variable z (2) bound by some $\mathsf{let}\ z = b_1\ \mathsf{in}\ b_2$ expression is instantiated by $\Phi(e_z)$ so as to transform the type of $[\![b_1]\!]$ into the type of this occurrence of z, according to the edge e_z; each occurrence of z in $[\![b_2]\!]$ will potentially pick a different instance. Thus, in the translation of $\mathsf{let}\ z = b_1\ \mathsf{in}\ b_2$ (3), the translation of b_1 is bound to z uninstantiated (as it suffices to instantiate the occurrences of z), while the translation of b_2 is instantiated according to the edge e_2. In the translation of an abstraction $\lambda(y)\ b$ (4), we annotate y by its type in the presolution (written $\mathcal{T}(y)$ and defined in §3.4) and coerce $[\![b]\!]$ to its type inside the abstraction according to the edge e. Finally, the translation of an application (5) is the application of the translations, each of which is instantiated according to its constraint edge.

Example 9. *The presolution χ_p in Figure 15 can be translated into the term*

$$\Lambda(\alpha)\ \Lambda(\beta \geqslant \forall\,(\delta)\ \delta \to \alpha)\ \lambda(x : \alpha)\ (\Lambda(\gamma)\ \lambda(y : \gamma)\ (x\ \mathbb{1}))\ (!\beta)$$

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 $!\beta$ is the translation of e.

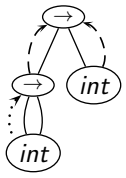
Type-erasure. As we will see later, $\bigwedge\!\!\backslash(g)$ is only composed of type quantifications, and $\Phi(e)$ of instantiations. Thus, the translation is type-erasure preserving by construction, which ensures that the semantics of the original and translated terms are the same.

Theorem 19. *Given a (desugared) term b, we have $\lceil [\![b]\!] \rceil = \lceil b \rceil$.*

3.3. Rigidifying presolutions

All presolutions are not suitable for elaboration into $x\mathsf{ML}^\mathsf{F}$, because rigid and flexible bindings are not treated symmetrically during the translation. Indeed, $x\mathsf{ML}^\mathsf{F}$ has flexible quantification, but does not have the rigid form. Rigid quantification is not necessary in $x\mathsf{ML}^\mathsf{F}$ because types are fully explicit and rigid nodes can always be explicitly unshared. Unsurprisingly, flexible bindings will be translated to flexible quantification—while rigid bindings will be inlined.

This causes a problem with inert nodes that are flexible but bound under a rigid edge: while they are instantiable in $e\mathsf{ML}^\mathsf{F}$ in any context, they would appear in a non instantiable context in $x\mathsf{ML}^\mathsf{F}$ if we translated them as flexible bounds, and there would be no way to instantiate them afterward. One solution is to inline them during the translation, exactly as rigid bounds. However, an even simpler solution is to rigidify them prior to the translation. This is a sound operation in $e\mathsf{ML}^\mathsf{F}$, since inert nodes can always be weakened, and it avoids a special case during the translation.



Example 10. *For example, the flexible binding edge in the type drawn on the right, which is leaving from the inert node* $\langle 11 \rangle$*, may be weakened in* $e\mathsf{ML}^\mathsf{F}$*. The two types with or without rigidification are equivalent in* $e\mathsf{ML}^\mathsf{F}$*. However, they are translated into* $(\forall\,(\alpha \geqslant \mathsf{int})\ \alpha \to \alpha) \to \mathsf{int}$ *and* $(\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$*, which are not equivalent in* $x\mathsf{ML}^\mathsf{F}$ *(in this case, they are actually incomparable): 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 types* $(\forall\,(\alpha)\ \mathsf{int} \to \mathsf{int}) \to \mathsf{int}$ *and* $(\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$ *in System F.*

For now, let us call *rigidification* the weakening of an inert node. A weakening is in general a strict instance operation in $e\mathsf{ML}^\mathsf{F}$. However, on inert nodes it is a *lossless* one as it right-commutes with all instance operations: a rigidification followed by an instantiation can always be rewritten as an instantiation followed by a rigidification. This means that rigidification will never make typechecking fail when it would not fail without rigidification. Intuitively, when an inert node n that has been rigidified is unified with another inert node m, then m itself can always be rigidified so that unification succeeds, because it is already or can be made inert.[8]

Although inert nodes in non-instantiable contexts are the only nodes that *must* be rigidified, all inert nodes *may* be rigidified. This is easier to implement, but more importantly, it results in simpler and more uniform elaborated terms.

[8] This reasoning can actually be generalized to lowering and splitting of inert nodes, which explains why we only need direct instance operations on such nodes.

For the same reason, we also rigidify flexible existential nodes, even though these are not inert. An existential node is bound to a G-node but not reachable by structure edges. If it is rigid, it will be inlined by the translation. But no occurrence will be found, so it will be skipped. However, if it is flexible, its translation introduces a (useless) type abstraction over a variable that does not appear in the body of the abstraction but that would still have to be eliminated by some irrelevant type application. Rigidifying flexible existential nodes is always correct and still lossless. Moreover, it avoids useless abstraction and applications in the translated term, as in Example 10.

Since presolutions are instances of the original type constrains (no node and no edge have been lost), we can describe rigidification on the typing constrains of Figure 14. Namely, the following nodes of the presolution are rigidified:

- the node $\langle g1 \rangle$ in the translation of abstractions (4);
- the node n in the translation of an application (5);
- the node $\langle g1 \rangle$ whenever it is bound on g;
- any node bound on a G-node but not reachable from a G-node by following only structure edges (*i.e.* an existential node).

In the first two cases, rigidification could have been performed during constraint generation since nodes that are rigidified are already inert in the constraint. Conversely, in the two last cases, it is important that the nodes are left flexible *during* type inference when some of the constraints might not have yet been solved, and rigidified only *after* type inference, *i.e.* in presolutions so that rigidification remains a lossless transformation, as argued earlier. Notice that although nodes $\langle g1 \rangle$ are always bound on $\langle g \rangle$ in the original constraint, they might be bound above in the presolution, in which case they must not be rigidified—unless they have been merged with other nodes that must be rigidified according to the criteria above.

We call *rigid* a presolution that respects the four conditions above and in which all inert nodes are rigid. We call *rigidification* the transformation of a presolution into a most general, rigid one. The following lemma states the existence of lossless rigid presolutions.

Lemma 20. *Given a presolution* χ_p *of a constraint* χ*, there exists a rigid presolution* χ'_p *of* χ*, derived from* χ_p *only by rigidifying some nodes, such that the solutions of* χ_p *and* χ'_p *are equivalent up to the weakening of inert nodes.*

This result suggests that we could have restricted ourselves to rigid presolutions 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

$$\mathcal{R}_\chi(n) \triangleq \forall(\mathcal{Q}_\chi(n))\ \chi(n)\ (\mathcal{T}_\chi(\langle n\,1\rangle), \dots \mathcal{T}_\chi(\langle n\,p\rangle))$$
where p is the arity of $\chi(n)$

$$\mathcal{T}_\chi(n) \triangleq \begin{cases}\mathcal{R}_\chi(n) & \text{if } n \text{ is rigidly bound in } \chi\\ \alpha_n & \text{if } n \text{ is flexibly bound in } \chi\end{cases}$$

$$\mathcal{Q}_\chi(n) \triangleq (\alpha_{\langle n_1\rangle} \geqslant \mathcal{R}_\chi(n_1) \dots \alpha_{\langle n_k\rangle} \geqslant \mathcal{R}_\chi(n_k))$$
where $n_1, \dots n_k$ are all non G-nodes flexibly bound to n in χ, ordered by $\prec$.

$$\mathcal{G}_\chi(g) \triangleq \forall(\mathcal{Q}_\chi(g))\ \mathcal{T}_\chi(\langle g\,1\rangle)$$

Figure 16: Mapping nodes of $e\mathsf{ML}^\mathsf{F}$ to types of $x\mathsf{ML}^\mathsf{F}$.

only be rigidified—without losing solutions—after all the constraint edges under them have been solved. This imposes some synchronization during the constraint resolution. Therefore, we prefer to stay with the more flexible (and simpler) definition of presolutions for $e\mathsf{ML}^\mathsf{F}$ and perform rigidification as a first step of the translation into $x\mathsf{ML}^\mathsf{F}$. This way, rigidification needs not be exposed to the user.

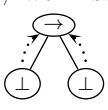
In the remainder of this section, we abstract over a rigid presolution χ and an instantiation edge e of the form $g \Longrightarrow d$.

3.4. Translating types

Ordering binders. 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, even if reordering of quantifiers remains necessary in some cases, as described below (§3.4, page 18).

The simplest way to order quantifiers is to assume a total ordering $\prec$ of all the nodes of a constraint χ. Of course, $\prec$ cannot be arbitrary, as it should also ensure the well-scopedness of syntactic types: if $n \circ\!\!\rightarrow n'$ or $n' \succ\!\!\rightarrow n$, then $n' \prec n$ must hold.

We choose the leftmost-lowermost ordering of nodes for $\prec$. That is, if $n_1, \dots, n_k$ are bound to n, we first translate the n_i that is structurally lowest in the type, or leftmost if the n_i are not ordered by $\circ\!\!\rightarrow$. This means that the type drawn on the right is always translated as $\forall(\alpha_1)\ \forall(\alpha_2)\ \alpha_1 \to \alpha_2$, not as $\forall(\alpha_2)\ \forall(\alpha_1)\ \alpha_1 \to \alpha_2$.



From graphic types to $x\mathsf{ML}^\mathsf{F}$ types. Every node of χ can be translated to an $x\mathsf{ML}^\mathsf{F}$ type. Moreover, the translation is uniquely determined by the ordering of binders.

We assume that every node n in χ is in bijection with a type variable α_n. Each non G-node n of χ is mapped to a type $\mathcal{T}_\chi(n)$ of $x\mathsf{ML}^\mathsf{F}$ as described in Figure 16. A flexibly bound node is translated by $\mathcal{T}_\chi$ as α_n; this translation is always used in a context where α_n is bound. Otherwise, n is rigidly bound and its type is inlined as $\mathcal{R}_\chi(n)$ whose definition uses a helper function $\mathcal{Q}_\chi(n)$ to build a sequence of type quantifications (one for each node flexibly bound to n); then $\mathcal{R}_\chi(n)$ is also used recursively to build the bounds of the type variables in $\mathcal{Q}(n)$. When χ is clear from context, we omit it for brevity.

Example 11. *Consider again Figure 12, disregarding the expanded part on the right for now. Let us consider the translation of the node $\langle n1\rangle$ (the arrow node under n). There is only one node bound on it, the node $\langle n12\rangle$, whose bound is $\bot$. Hence, $\mathcal{T}(\langle n1\rangle)$ is $\forall(\alpha_{\langle n12\rangle} \geqslant \bot)\ \alpha_{\langle n11\rangle} \to \alpha_{\langle n12\rangle}$.*

The function $\mathcal{G}$ is used to translate a G-node g. This is done by introducing the sequence of type quantifications $\mathcal{Q}(n)$ (representing the type variables generalized at the level of the type scheme that g stands for), followed by the translation of $\langle g\,1\rangle$. Notice that some other type quantifications can be introduced when translating $\langle g\,1\rangle$; this stands for polymorphism purely local to g. That is, this polymorphism was already present in g, has not been instantiated, and needs not be re-introduced. Notice also that, by definition of rigid presolutions, $\langle g\,1\rangle$ cannot be flexibly bound on g. Hence, the translation is never of the form $\forall(...)\ \forall(\alpha \geqslant \tau)\ \alpha$.

Finally, we write $\mathcal{G}(\chi)$ for the translation $\mathcal{G}(\langle\rangle)$ of the root G-node of the whole constraint.

Example 12 (cont.). *Let us focus on the root of the constraint in Figure 12. The non-G nodes that are flexibly bound on $\langle\rangle$ before expansion are $\langle 11\rangle$ and $\langle n11\rangle$. As n is also $\langle 12\rangle$, we have $\langle 11\rangle \prec \langle n11\rangle$. Thus, the translation $\mathcal{G}(\langle\rangle)$ of $\langle\rangle$ is*

$$\forall(\alpha_{\langle 11\rangle} \geqslant \bot)\ \forall(\alpha_{\langle n11\rangle} \geqslant \bot)\ \alpha_{\langle 11\rangle} \to (\mathcal{T}(\langle n1\rangle) \to \alpha_{\langle 11\rangle})$$

Given all these definitions, we are now able to formally define the notation $\bigwedge\!\!\!\backslash(g)$ used in Figure 14. It is simply $\Lambda(\mathcal{Q}(g))$.

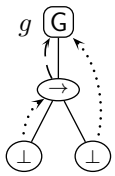
Translating the type of an expansion. Let χ be a constraint containing an instantiation edge e equal to $g \Longrightarrow d$. Let χ' be an instance of the expansion χ^e of e in χ, such that $\chi^e \sqsubseteq \chi' \sqsubseteq \chi$. Let r be the root of the expanded (*i.e.* copied) part in χ'. In §3.5, we will need to refer to the type under r, as we will transform this type so that it matches the type under d. It would be meaningless to translate r as $\alpha_{\langle r\rangle}$, because after any transformation under r, the translation would still be $\alpha_{\langle r\rangle}$. Instead, the correct type is the following: if r has been created by the expansion, we inline it regardless of its binding flag, and translate it as $\mathcal{R}_{\chi'}(r)$. Conversely, if r is in fact d, it is translated as $\mathcal{T}_{\chi'}(d)$ as usual.[9] Formally, the translation $\mathcal{E}_{\chi'}(r)$ of r is

[9] The case r equal to d happens either when χ' has been instantiated back into χ or when g is degenerate in χ and does not hold

defined as

$$\mathcal{E}_{\chi'}(r) \triangleq \begin{cases} \mathcal{T}_{\chi'}(r) & \text{if } r \text{ is } d \\ \mathcal{R}_{\chi'}(r) & \text{otherwise} \end{cases}$$

Example 13 (cont.). *In Figure 12, the translation of r is the type $\forall\,(\alpha_{\langle r1\rangle} \geqslant \bot)\ \alpha_{\langle r1\rangle} \to \alpha_{\langle 11\rangle}$, as r is not part of the initial constraint.*



Type of a G *node vs. type of an expansion.* In some cases, the translation of the expansion does not correspond to the translation of g, regardless of our use of $\prec$. This can easily be seen in the example drawn on the right. Here $\mathcal{G}(g)$ 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, the two translations may differ only by a reordering of quantifiers. In $x\mathsf{ML}^\mathsf{F}$, we can explicitly reorder them using the instantiation

$$\text{⅋}; \forall\,(\geqslant @\tau_\alpha);\ \text{⅋}; \forall\,(\geqslant @\tau_\beta);\ \forall\,(\beta \geqslant)\ \forall\,(\alpha \geqslant)\ \big(\langle !\alpha\rangle;\ \langle !\beta\rangle\big)$$

whose effect is just to commute α and β in the type $\forall\,(\alpha \geqslant \tau_\alpha)\ \forall\,(\beta \geqslant \tau_\beta)\ \tau$.

In the general case, we write $\Sigma(g)$ for the instantiation that transforms $\mathcal{G}(g)$ into the translation of its expansion.

3.5. From instantiation edges to type instantiations

The main part of the translation is the computation of the type instantiation $\Phi(e)$ corresponding to an instantiation edge e. By assumption, the edge is solved; thus χ is an instance of the expansion χ^e of e in χ. This instantiation can be witnessed by a sequence Ω of atomic instance operations. We first build a graphical instantiation Ω that will then be translated into a type instantiation in $x\mathsf{ML}^\mathsf{F}$.

Building Ω. Because Ω leaves χ unchanged (as otherwise $\chi^e \sqsubseteq \chi$ would not hold, $\sqsubseteq$ being antisymmetric), the operations can be rearranged into the following forms (we let r be the root node of the expansion in χ^e):

(**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 $n \succ\!\!\stackrel{+}{\to}\!\!\succ\!\!\longrightarrow r$;
(**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. Conversely, the 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 $\mathcal{I}(r)$.

Furthermore, since χ is a rigid presolution, we may also assume that an operation $\mathsf{Weaken}(n)$ appears after all the other operations on a node below n (**5**). 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 by Yakobowski (2008) and it is actually quite easy to establish: performing all instance operations bottom-up, while delaying weakening operations as much as possible, is always possible and results in a normalized witness.

Example 14. *The constraint edge e of χ in Figure 13 is solved. We recall the witness of $\chi^e \sqsubseteq \chi$ that we gave in Example 7:*

$\mathsf{Graft}(\forall\,(\alpha)\ \forall\,(\beta)\ \alpha \to \beta, \langle r1\rangle)$
$\mathsf{Raise}(\langle r11\rangle)$; $\mathsf{Raise}(\langle r11\rangle)$; $\mathsf{Merge}(\langle r11\rangle, \langle n11\rangle)$;
$\mathsf{Weaken}(\langle r1\rangle)$; $\mathsf{Weaken}(r)$; $\mathsf{Merge}(r, n)$

This transformation is not normalized because node $\langle r11\rangle$ is raised twice above the root r, then merged with $\langle n11\rangle$. We must join those three operations into $\mathsf{RaiseMerge}(\langle r11\rangle, \langle n11\rangle)$. Similarly, the last operation merges n and r and should be replaced by $\mathsf{RaiseMerge}(r, n)$. This results in the following normalized derivation:

$\mathsf{Graft}(\forall\,(\alpha)\ \forall\,(\beta)\ \alpha \to \beta, \langle r1\rangle)$;
$\mathsf{RaiseMerge}(\langle r11\rangle, \langle n11\rangle)$;
$\mathsf{Weaken}(\langle r1\rangle)$; $\mathsf{Weaken}(r)$; $\mathsf{RaiseMerge}(r, n)$

Similarly, in Figure 15, we have $\chi^e_p \sqsubseteq \chi_p$ —as witnessed by $\mathsf{RaiseMerge}(r, n)$, which is normalized, hence equal to $\Omega(e)$.

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 flexible node m that is 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 $\mathcal{R}(n)$. For presolutions, in order to avoid α-conversion-related issues, we build instantiation contexts using variables whose names are based on the nodes they traverse.

polymorphism; see, *e.g.*, the lowermost G-node in Figure 15 in which case both r and d are equal to $\langle g\,1\rangle$ in χ^e.

Any operation on a node that is transitively bound to the root of an expansion can be expressed using an instantiation context (and a "local" instantiation). Conversely, the operations on rigidly bound or inert-locked nodes cannot. This is unimportant in our case, as normalized witnesses of rigid presolutions only transform nodes transitively flexibly bound to the root of the expansion.

Example 15. *For example, consider the constraint χ_p in Figure 15. The translation $\mathcal{Q}(\langle\rangle)$ of the root* G*-node is*

$$\forall\,(\alpha_{\langle 11\rangle}\geqslant\bot)\;\forall\,(\alpha_{\langle 12\rangle}\geqslant\forall\,(\alpha_{\langle 121\rangle}\geqslant\bot)\;\alpha_{\langle 121\rangle}\to\alpha_{\langle 11\rangle})\;\alpha_{\langle 11\rangle}\to\alpha_{\langle 12\rangle}$$

With the convention above, $\mathcal{C}^{\langle\rangle}_{\langle 11\rangle}=\{\cdot\}$, $\mathcal{C}^{\langle\rangle}_{\langle 12\rangle}=\forall\,(\alpha_{\langle 11\rangle}\geqslant)\,\{\cdot\}$, and $\mathcal{C}^{\langle\rangle}_{\langle 121\rangle}=\forall\,(\alpha_{\langle 11\rangle}\geqslant)\,\forall\,(\geqslant\{\cdot\})$.

Translating normalized derivations into instantiations. Let us resume the construction of $\Phi(e)$ by translating a normalized witness Ω of $\chi^e\sqsubseteq\chi$ into a type instantiation in $x\mathsf{ML}^\mathsf{F}$. In fact, we generalize the problem by translating a normalized witness Ω of $\xi\sqsubseteq\chi$ where ξ is an instance of χ^e, *i.e.* 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). We remind that $\mathcal{E}_\chi(r)$ is the translation of r in the constraint ξ. By definition, the translation of Ω, written $\Phi_\xi(\Omega)$, must witness the instantiation $\mathcal{E}_\xi(r)\leq\mathcal{E}_\chi(r)$, *i.e.*

$$\Gamma_d\vdash\Phi_\xi(\Omega):\mathcal{E}_\xi(r)\leq\mathcal{E}_\chi(r)$$

where Γ_d is the typing context for the node d.[10] The translation of Ω is defined by induction on Ω as described in Figure 17. 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 cannot be transformed at all. 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 $(@\tau)$, where τ is the translation of σ into $x\mathsf{ML}^\mathsf{F}$. (Grafting grafts only closed types, so the constraint in which we translate σ is unimportant.)
- A raise-merge of r with m is translated to $!\alpha_m$: it must be the last operation of the derivation Ω, and α_m is necessarily bound in the typing environment Γ_d; hence we may abstract the type of r under α_m.

[10] We do not define the typing contexts Γ_d formally, since they are not needed for the translation, but only to state its properties.

Sequences of operations

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

Operation ω on a rigid node n

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

Operation on the (flexible) root r of the expansion

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

Operation on a flexible node different from the root

$$\begin{aligned}\Phi_\xi(\mathsf{Graft}(\sigma,n)) &= \mathcal{C}^r_n\{\forall\,(\geqslant@(\mathcal{R}(\sigma)))\}\\ \Phi_\xi(\mathsf{RaiseMerge}(n,m)) &= \mathcal{C}^r_n\{\forall\,(\geqslant!\alpha_m);\&\}\\ \Phi_\xi(\mathsf{Merge}(n,m)) &= \mathcal{C}^r_n\{\forall\,(\geqslant!\alpha_m);\&\}\\ \Phi_\xi(\mathsf{Weaken}(n)) &= \mathcal{C}^r_n\{\&\}\\ \Phi_\xi(\mathsf{Raise}(n)) &= \mathcal{C}^r_m\{\wp;\forall\,(\geqslant@(\mathcal{R}_\xi(n)));\\ &\qquad\forall\,(\beta_n\geqslant)\;\mathcal{C}^m_n\{\forall\,(\geqslant!\beta_n);\&\}\}\end{aligned}$$

where $m=\min_\prec\{m\mid n\succ\!\!\rightarrow\!\!\succ\!\!\rightarrow\!\!\leftarrow\!\!\prec m\wedge n\prec m\}$

Figure 17: Translating normalized instance operations

- 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.

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 flexible and transitively bound to r. Therefore, there exists an instantiation context $\mathcal{C}^r_n$, which we call $\mathcal{C}$, to reach the bound of α_n in $\mathcal{R}_\xi(r)$.

- The grafting of a type σ at n is translated to $\mathcal{C}\{\forall\,(\geqslant@(\mathcal{R}(\sigma)))\}$ which transforms the bound $\bot$ of α_n into $\mathcal{R}(\sigma)$.
- The merging of n with a node m is translated to $\mathcal{C}\{\langle!\alpha_m\rangle\}$, which first abstracts the bound of α_n under the name α_m and immediately eliminates the quantification. (We have assumed $m\prec n$, hence α_m is in scope in the bound of n.)
- The translation is the same for a raise-merge, but α_m is bound in the typing environment instead of in $\mathcal{R}_\xi(r)$.
- The weakening of n is translated to $\mathcal{C}\{\&\}$, which eliminates the bound of n.
- Finally, the translation of the raising of n is more involved, and of the form $\mathcal{C}^r_m\{\wp;\forall\,(\geqslant@(\mathcal{R}_\xi(n)));\phi\}$.

  We first insert a fresh quantification, which will be the one of n after the raising, inside $\mathcal{R}_\xi(r)$. The

bound is the current bound of n, *i.e.* $\mathcal{R}_\xi(n)$. 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 $\{m \mid n \succ\!\!\rightarrow \succ\!\!\rightarrow \prec\!\!\leftarrow\!\!\prec m \wedge n \prec m\}$ contains at least the binder of n, hence its minimum m is well-defined. Then, the instantiation ϕ equal to $\forall\,(\beta_n \geqslant)\ \mathcal{C}_n^m\{\forall\,(\geqslant !\beta_n); \&\}$ 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.

Finally, in order to have a correct instantiation, it remains to reorder quantifiers as described earlier (page 18). Thus we take

$$\Phi(e) = \Sigma(g); \Phi_{\chi^e}(\Omega)$$

Example 16 (cont.). *The translation of each step of the normalized witness of Example 14 is:*

Normalized graphic operation	*xML^F translation*
$\mathsf{Graft}(\forall\,(\alpha)\ \forall\,(\beta)\ \alpha \to \beta, \langle r1\rangle)$	$\forall\,(\geqslant @(\mathcal{R}(\forall\,(\alpha)\ \forall\,(\beta)\ \alpha \to \beta)))$
$\mathsf{RaiseMerge}(\langle r11\rangle, \langle n11\rangle)$	$\forall\,(\geqslant \forall\,(\geqslant !\alpha_{\langle n11\rangle}))$
$\mathsf{Weaken}(\langle r1\rangle)$	$\&$
$\mathsf{Weaken}(\langle r\rangle)$	$\mathbb{1}$
$\mathsf{RaiseMerge}(\langle r\rangle, \langle n\rangle)$	$!\alpha_n$

Since for the edge e of χ we have $\Sigma(g) = \mathbb{1}$, the entire translation of e is

$$\Phi(e) = \mathbb{1}; \forall\,(\geqslant @(\mathcal{R}(\forall\,(\alpha)\ \forall\,(\beta)\ \alpha \to \beta))); \forall\,(\geqslant \forall\,(\geqslant !\alpha_{\langle n11\rangle})); \&; \mathbb{1}; \mathbb{1}$$

3.6. Translating annotated terms

As mentioned in §3.1.3, 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 $\mathcal{R}(\kappa_\sigma)$ of the type of the coercion function κ_σ in $x\mathsf{ML}^\mathsf{F}$ is $\forall\,(\alpha \geqslant \mathcal{R}(\sigma))\ \mathcal{R}(\sigma) \to \alpha$. Interestingly, coercion functions need not be primitive in $x\mathsf{ML}^\mathsf{F}$—unlike in $e\mathsf{ML}^\mathsf{F}$. Let id_κ be the expression $\Lambda(\alpha)\ \Lambda(\beta \geqslant \alpha)\ \lambda(x : \alpha)\ (x\ (!\beta))$. Then, define κ_σ as $\mathsf{id}_\kappa\langle\mathcal{R}(\sigma)\rangle$. Notice that κ_σ behaves as the identity function. Moreover, coercion functions can always be eliminated by strong reduction (as implied by Lemma 13) in the elaboration of the presolution, so that they have no runtime cost.

3.7. Soundness of the translation

Theorem 21. *Let b be an eML^F term, χ a rigid presolution for b. The translation $[\![\, b \,]\!]$ of χ is well-typed in xML^F, of type $\mathcal{G}(\chi)$.*

Our translation preserves the type-erasure of programs (Theorem 19). Hence, the soundness of $x\mathsf{ML}^\mathsf{F}$ also implies the soundness of $e\mathsf{ML}^\mathsf{F}$—which had previously only been proved for the syntactic versions of ML^F, but not for the most general, graphical version.

3.8. Optimizations

The elaboration is a compilation process, and we have defined it in its simplest form. In practice, some optimizations could be performed during the elaboration process. For instance, raising k times a node n (to a position n'), is currently done step by step by invoking the atomic $\mathsf{Raise}(n)$ operation k times. This could (and should) be translated in a simple step, avoiding intermediate abstractions and applications in $x\mathsf{ML}^\mathsf{F}$. Similarly, contexts could be factored, replacing $\mathcal{C}_n^r(\phi); \mathcal{C}_n^r(\phi')$ by $\mathcal{C}_n^r(\phi; \phi')$. Those optimizations are actually straightforward and significantly simplify elaborated terms—they have been implemented in our prototype (Scherer, 2010b), indeed. Optimizations can also be performed a posteriori, by transforming $x\mathsf{ML}^\mathsf{F}$ terms into equivalent ones (with the same type and the same type erasure), as discussed in §5.2.

4. Expressiveness of $x\mathsf{ML}^\mathsf{F}$

The translation of $e\mathsf{ML}^\mathsf{F}$ into $x\mathsf{ML}^\mathsf{F}$ shows that $x\mathsf{ML}^\mathsf{F}$ is at least as expressive as $e\mathsf{ML}^\mathsf{F}$. However, the converse is not true. (This is not entirely surprising: as mentioned in §3.6, coercion functions are primitive in $e\mathsf{ML}^\mathsf{F}$, but not in $x\mathsf{ML}^\mathsf{F}$.) That is, there exist programs of $x\mathsf{ML}^\mathsf{F}$ that cannot be typed in $e\mathsf{ML}^\mathsf{F}$. While this is mostly irrelevant when using $x\mathsf{ML}^\mathsf{F}$ as an internal language, the question is still interesting from a theoretical point of view, and may help understanding ML^F independently of any restriction imposed for the purpose of type inference and perhaps suggest other useful extensions.

For the sake of simplicity, we explain the difference between $x\mathsf{ML}^\mathsf{F}$ and $i\mathsf{ML}^\mathsf{F}$, the Curry-style version of ML^F (which has the same expressiveness as $e\mathsf{ML}^\mathsf{F}$, but does not require explicit type annotations in source terms).

4.1. A term typable in xML^F but not in iML^F

Although syntactically identical, the types of $x\mathsf{ML}^\mathsf{F}$ and of syntactic $i\mathsf{ML}^\mathsf{F}$ differ in their interpretation of 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 $\forall\,(\alpha \geqslant \tau)\ \alpha \to \alpha$ respectively. In $i\mathsf{ML}^\mathsf{F}$, β is just an alias for α and these two types are equivalent. Intuitively, the set of their instances (stripped of toplevel quantifiers) is $\{\tau' \to \tau' \mid \tau \leq \tau'\}$. In $x\mathsf{ML}^\mathsf{F}$, the set of instances of τ_0 is larger and at least a superset of $\{\tau'' \to \tau' \mid \tau \leq \tau' \leq \tau''\}$, which can be obtained from τ_{id} by all type instantiations of the form $\forall\,(\geqslant \phi); \&; \forall\,(\geqslant \phi'); \&$ with $\vdash \phi : \tau \leq \tau'$ and $\vdash \phi' : \tau' \leq \tau''$. That is, an instance of τ_0 can pick for β an instance of the type chosen for α. This level of generality, possible in $x\mathsf{ML}^\mathsf{F}$, cannot be expressed in $i\mathsf{ML}^\mathsf{F}$.

From this observation, we may easily exhibit an expression a that is typable in $x\mathsf{ML}^\mathsf{F}$ but not in $i\mathsf{ML}^\mathsf{F}$. For readability of the example, we assume primitive products. Let a_0 be the expression

$$\Lambda(\alpha)\ \Lambda(\beta \geqslant \alpha)\ \lambda(x : \alpha)\ \lambda(y : \beta)\ (x, \mathsf{choice}\ \langle\beta\rangle\ (x\ (!\beta))\ y)$$

$$
\begin{array}{rcl}
[\![\lambda(x:\tau)\ a]\!]_\Delta & = & \lambda(x:\exists(\Delta)\ \tau)\ [\![a]\!]_\Delta \\
 & \triangleq & \lambda(x)\ \mathsf{let}\ x = (\exists(\Delta)\ \tau)\ x\ \mathsf{in}\ [\![a]\!]_\Delta \\
[\![x]\!]_\Delta & = & x \\
[\![a_1\ a_2]\!]_\Delta & = & [\![a_1]\!]_\Delta\ [\![a_2]\!]_\Delta \\
[\![\Lambda(\alpha \geqslant \rho)\ a]\!]_\Delta & = & [\![a]\!]_{\Delta,\alpha\geqslant\rho} \\
[\![a\ \phi]\!]_\Delta & = & [\![a]\!]_\Delta
\end{array}
$$

Figure 18: Translating $x\mathsf{ML}^\mathsf{F}$ into $e\mathsf{ML}^\mathsf{F}$

of type $\tau_0 \triangleq \forall(\alpha)\,\forall(\beta \geqslant \alpha)\ \alpha \to \beta \to (\alpha \times \beta)$. Let a_1 and a_2 be defined as

$$
\begin{array}{rcl}
a_1 & \triangleq & \Lambda(\alpha)\ \lambda(x:\alpha)\ x \qquad : \qquad \forall(\alpha)\ \alpha \to \alpha \ \triangleq\ \tau_1 \\
a_2 & \triangleq & \Lambda(\alpha)\ \lambda(x:\alpha)\ \lambda(y:\alpha)\ x : \forall(\alpha)\ \alpha \to \alpha \to \alpha \ \triangleq\ \tau_2
\end{array}
$$

Let i be 1 or 2 and a'_i be $\lambda(x:\tau_i)\ x\ \langle\tau_i\rangle\ x$. We have $\vdash a'_i : \tau'_i$, where τ'_i is defined as $\tau_i\ \langle\tau_i\rangle$. If f has type τ_0, then $f\ (\langle\sigma\rangle; \forall(\geqslant\phi); \&)$ has type $\sigma \to \sigma' \to (\sigma \times \sigma')$, for any instantiation ϕ such that $\phi \vdash \sigma \leq \sigma'$. Let ϕ_i be $\langle\tau_i\rangle; \forall(\geqslant\langle\tau_i\rangle); \&$ and τ''_i be $\tau_i \to \tau'_i \to (\tau_i \times \tau'_i)$ and observe that $\phi_i \vdash \tau_0 \leq \tau''_i$. Let a''_i be $\mathsf{let}\ (x_i, x'_i) = f\ \phi_i\ a_i\ a'_i\ \mathsf{in}\ x'_i\ x_i$ and take $(\lambda(f:\tau_0)\ (a''_1, a''_2))\ a_0$ for a. The expression a is well-typed in $x\mathsf{ML}^\mathsf{F}$ (and has type $\tau_1 \times (\tau_2 \to \tau_2)$).

However, the type erasure of a is ill-typed in $i\mathsf{ML}^\mathsf{F}$, as there is no annotation τ_0 for the type of the parameter f that is simultaneously a correct type for $\lceil a_0 \rceil$ and that can be independently instantiated to τ''_1 and τ''_2—or some other types that allow to simultaneously type both expressions a''_1 and a''_2. The problem is that, in $i\mathsf{ML}^\mathsf{F}$, $\lceil a_0 \rceil$ can only be given a type of the form $\tau \to \tau' \to (\tau \times \tau'')$ or $\tau' \to \tau \to (\tau' \times \tau'')$ with $\tau \leq \tau' \leq \tau''$, or of the form $\forall(\alpha)\ \alpha \to \alpha \to (\alpha \times \alpha)$ (in which both arguments must have identical types), but not simultaneously two such types.

4.2. Restricting $x\mathsf{ML}^\mathsf{F}$ to match $e\mathsf{ML}^\mathsf{F}$

The current treatment of variable bounds in $x\mathsf{ML}^\mathsf{F}$ is quite natural in a Church-style presentation. Surprisingly, it is also simpler than treating them as in $e\mathsf{ML}^\mathsf{F}$. A restriction $x\mathsf{ML}^\mathsf{F}_\flat$ of $x\mathsf{ML}^\mathsf{F}$ without variable bounds that is closed under reduction and in close correspondence with $i\mathsf{ML}^\mathsf{F}$ can still be defined a posteriori, by constraining the formation of terms. But the definition is contrived and unnatural, and may not be so appealing in practice (see Appendix C for details). Still, all terms of $x\mathsf{ML}^\mathsf{F}_\flat$ can be translated to $e\mathsf{ML}^\mathsf{F}$.

The translation is actually very similar to that for Church-style System F (Le Botlan and Rémy, 2009) and proceeds by dropping all type abstractions and type applications and translating type annotations of argument of functions. As a result, some type variable may become free in translated types and must be existentially quantified, leading to annotations of the form $\exists(\Delta)\ \tau$. Free variables are kept with their bound in the source. Hence, Δ is a list of $\alpha_i \geqslant \rho_i$ where ρ are non-variable types (see Appendix C). This is a minor difference with System F where all bounds are trivial—and thus need not be tracked. Here, the translation uses an environment to pass this information downward as described in Figure 18. The annotation $\exists(\Delta)\ \tau$ stands in $e\mathsf{ML}^\mathsf{F}$ for the coercion function of type $\forall(\Delta)\ \forall(\alpha = \tau)\ \forall(\alpha' = \tau)\ \alpha \to \alpha'$, which can easily be translated into some graphic type, as described in (Yakobowski, 2008, Chapter 8).

The restriction to $x\mathsf{ML}^\mathsf{F}_\flat$ prevents the use of variable bounds and therefore of type instantiation between types whose translation into $e\mathsf{ML}^\mathsf{F}$ would not be in some instance relationship. This should ensure that the translation of well-typed terms is well-typed, although we have not checked it formally.

Notice that the translation described above annotates all parameters of functions, which is not necessary in $e\mathsf{ML}^\mathsf{F}$. Only parameters of functions that are used polymorphically need to be annotated. A simple optimization is to omit monomorphic type annotations, *i.e.* type annotations of the form $\exists(\Delta)\ \tau$ where neither Δ nor τ contain quantifiers. Still all parameters of functions that have a polymorphic type, whether or not used polymorphically, will be annotated. The image of the translation is then in HML (Leijen, 2008), a strict subset of $e\mathsf{ML}^\mathsf{F}$. Indeed, parameters of functions that are polymorphic may still not be used polymorphically and need not be annotated in ML^F. However, we do not know whether this can be easily checked during the translation. (In fact, this would amount to detecting and removing useless type-annotations in $e\mathsf{ML}^\mathsf{F}$.)

4.3. Enriching $e\mathsf{ML}^\mathsf{F}$ to match $x\mathsf{ML}^\mathsf{F}$?

Instead of restricting $x\mathsf{ML}^\mathsf{F}$ to match the expressiveness of $i\mathsf{ML}^\mathsf{F}$, a question worth further investigation is whether the treatment of variable bounds could be enhanced in $i\mathsf{ML}^\mathsf{F}$ and $e\mathsf{ML}^\mathsf{F}$ to match their interpretation in $x\mathsf{ML}^\mathsf{F}$ but without compromising type inference. A solution might exist, but it would likely depart from $e\mathsf{ML}^\mathsf{F}$: graphic types have been introduced to simplify the metatheory of the syntactic presentation of ML^F and one of the simplifications was precisely to disallow variable bounds, which could be written in the syntactic presentation but lead to many complications.

4.4. Comparing $x\mathsf{ML}^\mathsf{F}$ and F^η

Type instantiation in $x\mathsf{ML}^\mathsf{F}$, which changes the type of an expression without changing its meaning, can be applied deeply inside a type while it is only superficial in System F. This has some resemblance with retyping functions in F^η, the closure of System F by η-conversion (Mitchell, 1988), which also allows deep type instantiations. However, type instantiations rely on quite different mechanisms in both languages. While it is explicitly expressed in flexible bounds in $x\mathsf{ML}^\mathsf{F}$, it is left implicit and driven by the underlying structure of types in F^η, propagating type instantiation covariantly on the right-hand side of arrow types and contravariantly on their left-hand side.

Both F^η and $x\mathsf{ML}^\mathsf{F}$ have a little more than System F in common, as our running example $\mathsf{choice\ id}$ has both types $\forall\,(\alpha)\ (\alpha \to \alpha) \to \alpha \to \alpha$ and $(\forall\,(\alpha)\ \alpha \to \alpha) \to (\forall\,(\alpha)\ \alpha \to \alpha)$, since the latter can be recovered from the former by type containment, distributing the $\forall$ over the arrow type constructor.

However, F^η fails on the application, $\mathsf{choice\ (choice\ id)}$: which is a small variant of $\mathsf{choice\ id}$: this program has type $\forall\,(\gamma \geqslant \forall\,(\beta \geqslant \forall\,(\alpha)\ \alpha \to \alpha)\ \beta \to \beta)\ \gamma \to \gamma$ in ML^F, which admits the three following particular System-F types as as instances:

$$((\forall\,(\alpha)\ \alpha \to \alpha) \to \forall\,(\alpha)\ \alpha \to \alpha) \to (\forall\,(\alpha)\ \alpha \to \alpha) \to \forall\,(\alpha)\ \alpha \to \alpha$$
$$(\forall\,(\alpha)\ (\alpha \to \alpha) \to \alpha \to \alpha) \to \forall\,(\alpha)\ (\alpha \to \alpha) \to \alpha \to \alpha$$
$$\forall\,(\alpha)\ ((\alpha \to \alpha) \to \alpha \to \alpha) \to (\alpha \to \alpha) \to \alpha \to \alpha$$

However, $\mathsf{choice\ (choice\ id)}$ does not have any type in F^η of which all these three types are instances.

Conversely, a function of type $\forall\,(\beta)\ (\tau_1\{\alpha \leftarrow \tau_2\} \to \beta) \to \beta$ can be seen as one of type $\forall\,(\beta)\ (\forall\,(\alpha)\ \tau_1 \to \beta) \to \beta$ in F^η by contra-variant type instantiation, which cannot (in general) be expressed $x\mathsf{ML}^\mathsf{F}$.

In fact $x\mathsf{ML}^\mathsf{F}$ and F^η are two rather orthogonal extensions of System F, which could be combined together, as shown in recent work by Cretin and Rémy (2012).

5. Discussion

5.1. Related works

A strong difference between $e\mathsf{ML}^\mathsf{F}$ and $x\mathsf{ML}^\mathsf{F}$ is the use of explicit coercions to trace the derivation of type instantiation judgments. Beside the several papers that describe variants of ML^F and are only indirectly related to this work, most related works are about the use of coercion functions in different ways.

Elaboration of ML^F into System F. In a way, the closest work to ours is the elaboration of ML^F into System F, first proposed by Leijen and Löh (2005) to extend ML^F with qualified types and later simplified by Leijen (2007) in the absence of qualified types. Since System F is less expressive than ML^F, an ML^F term a with a polymorphic type of the form $\forall\,(\alpha \geqslant \tau')\ \tau$ is elaborated as a function of type $\forall\,(\alpha)\ (\tau'_\star \to \alpha) \to \tau_\star$, where $\tau_\star$ is a runtime representation of τ. The first argument is a *runtime coercion*, which bears strong similarities with our instantiations. However, an important difference is that their coercions are at the level of terms, while our instantiations are at the level of types. In particular, although coercion functions should not change the semantics, this critical result has not been proved, and it is not obvious for a call-by-value language with side effects. In our setting the type-erasure semantics comes for free by construction.

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 proposed by Leijen (2007), in which rigid bindings are inlined during the translation. Both elaborations use some canonical ordering of quantifiers inside types, with slight differences: while we strive to reduce the number of quantifier reorderings, thus order all the quantifiers, Leijen uses only weaker canonical forms that are sufficient to obtain well-typed terms, but may result in additional reorderings.

Explicit coercions. A similar approach has already been used in a language with subtyping and intersection types, proposed as a target for the compilation of bounded polymorphism by Crary (2000). In both cases, coercions are used to make typechecking a trivial process. In our case, they are also exploited to make subject reduction easy—by introducing the language to describe how type instance derivations must be transformed during reduction. We believe that, more generally, the use of explicit coercions is a powerful tool for simplifying subject-reduction proofs. In both approaches, reduction is split into a standard notion of β-reduction and a new form of reduction (which we call ι-reduction) that only deals with coercions, preserves type-erasures, and is strongly normalizing. There are also important differences. While both coercion languages have common forms, our coercions intendedly keep the instance-bounded polymorphism form $\forall\,(\alpha \geqslant \tau)\ \tau'$. On the opposite, Crary uses the coercions to eliminate the subtype-bounded polymorphism form $\forall\,(\alpha \leq \tau)\ \tau'$, using intersection types and contravariant arrow coercions instead, which we do not need. Perhaps union types, which Crary (2000) proposes as an extension, could be used to encode away our instance-bounded polymorphism form.

Harnessing ML^F. In a recent paper, Manzonetto and Tranquilli (2010) have shown that $x\mathsf{ML}^\mathsf{F}$ is strongly normalizing by translation into System F, reusing the idea of Leijen and Löh (2005) and their translation of types, recalled above, but starting with $x\mathsf{ML}^\mathsf{F}$ instead of ML^F. It is unsurprising that the elaboration of ML^F into System F can be decomposed into our elaboration of ML^F into $x\mathsf{ML}^\mathsf{F}$ followed by a translation of $x\mathsf{ML}^\mathsf{F}$ into System F. However, the idea of Manzonetto and Tranquilli (2010) is to use the elaboration into System F to prove termination of the reduction in $x\mathsf{ML}^\mathsf{F}$ in some indirect but simple way, while a direct proof of termination seemed trickier. They show that the elaboration preserves well-typedness and the dynamic semantics via a simulation between the reduction of source terms and target terms. In this process, they also exhibit an intermediate calculus Fc of term-level retyping functions that mimic our type instantiations. Unfortunately, subject reduction does not hold in Fc (hence, we can only reuse their direct proof of bisimulation given in Appendix B). Moreover, their intermediate calculus Fc is tuned to be the target of $x\mathsf{ML}^\mathsf{F}$, and cannot express much more. It is actually subsumed by a calculus of erasable coercions F_ι recently proposed by Cretin and Rémy (2012), which contrary to Fc, enjoys subject reduction. Theorem 10 and Lemma 13

have also been verified by a translation of $x\mathsf{ML}^\mathsf{F}$ into F_ι (Cretin and Rémy, 2012).

System with type equalities. An extension of System F with type equality coercions, called FC or FC_2 for its revised version (Sulzmann et al., 2007; Weirich et al., 2011) has been proposed to be used as an internal language for Haskell. Type equalities are made explicit through witnesses that have some similarities with our instantiations. System FC has also been designed to be a compiler intermediate language, one of the objectives we have pursued with $x\mathsf{ML}^\mathsf{F}$. However, there are also significant differences: type coercions in FC_2 are type equality coercions while they are type instantiations in ML^F. In fact FC_2 is more related to the system F_ι mentioned above, of which $x\mathsf{ML}^\mathsf{F}$ is only a particular case. Technically, FC_2 and $x\mathsf{ML}^\mathsf{F}$ seems to be orthogonal extensions of System F which, perhaps, could be combined together. Unfortunately, ML^F-style polymorphism has been removed from the recent versions of GHC to better accommodate for type inference with GADT. We hope that this is temporary and that both could be eventually recombined.

5.2. Future works

The demand for an internal language for ML^F was first made in the context of using the $e\mathsf{ML}^\mathsf{F}$ type system for the Haskell language. We expect $x\mathsf{ML}^\mathsf{F}$ to better accommodate qualified types than $e\mathsf{ML}^\mathsf{F}$, since no evidence function should be needed for flexible polymorphism, but it 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 undoes 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 variable 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. In fact, the problem of choosing the right interpretation for variable bounds reappeared in a recent work by Scherer (2010a) on extending ML^F to cope with higher-order polymorphism. Indeed, this requires making coexist both implicit and explicit quantifiers, and using the $x\mathsf{ML}^\mathsf{F}$ interpretation for explicit quantifiers while retaining the ML^F more restrictive interpretation for implicit quantifiers.

As noticed in §4.4, type instantiation changes the type of an expression without changing its meaning. It can be performed deeply inside terms, as retyping functions in System F^η. In System F^η, retyping functions can be seen either at the level of terms, as expressions of System F that $\beta\eta$-reduce to the identity, or at the level of types as a *type conversion*. In $x\mathsf{ML}^\mathsf{F}$, retyping functions are at the level of types. However, the translation of type instantiations back into coercion functions as done by Manzonetto and Tranquilli (2010) allows one to also see them at the level of terms, bringing $x\mathsf{ML}^\mathsf{F}$ and F^η even closer. While the two languages differ in their coercions, they can be combined together as shown in recent work by Cretin and Rémy (2012), allowing a form of abstraction (as in $x\mathsf{ML}^\mathsf{F}$) over retyping functions (as in F^η).

Regarding type soundness, it is 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 (but only transformed) during reduction. Because $x\mathsf{ML}^\mathsf{F}$ has full type information, it cannot say anything about type information that could be left implicit and inferred. Given a term in $x\mathsf{ML}^\mathsf{F}$, can we rebuild a term in $i\mathsf{ML}^\mathsf{F}$ with minimal type annotations? While this should be easy if we require that corresponding subterms have identical types in $x\mathsf{ML}^\mathsf{F}$ and $i\mathsf{ML}^\mathsf{F}$, the answer is unclear if we allow subterms to have different types.

The semantics of $x\mathsf{ML}^\mathsf{F}$ allows reduction (and elimination) of type instantiations $a\ \phi$ through ι-reduction but does not allow reduction (and simplification) of instantiations ϕ alone. It would be possible to define a notion of reduction on instantiations $\phi \longrightarrow \phi'$ (such that $\forall(\geqslant\phi_1;\phi_2) \longrightarrow \forall(\geqslant\phi_1);\forall(\geqslant\phi_2)$, 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 type instantiations. 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; this 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 $e\mathsf{ML}^\mathsf{F}$ presolutions into $x\mathsf{ML}^\mathsf{F}$, by carefully selecting the instantiation witness to translate—as there usually exist more than one witness for a given instantiation edge. This amounts to using type derivations equivalence in $e\mathsf{ML}^\mathsf{F}$ instead of observational equivalence in $x\mathsf{ML}^\mathsf{F}$.

Extending ML^F with higher-order polymorphism is another ongoing research direction (Herms, 2009; Scherer, 2010a).

Conclusion

The Church-style version of $x\mathsf{ML}^\mathsf{F}$ that was still missing for type-aware compilation and from a theoretical point of view, completes the ML^F trilogy. The original type-inference version $e\mathsf{ML}^\mathsf{F}$, which requires partial type annotations but does not tell how to track them during reduction, now lies between the Curry-style presentation $i\mathsf{ML}^\mathsf{F}$ that ignores all type information and $x\mathsf{ML}^\mathsf{F}$ that maintains full type information during reduction.

We have shown that $x\mathsf{ML}^\mathsf{F}$ is well-behaved: reduction preserves well-typedness, and the calculus is sound for both call-by-value and call-by-name semantics.

Hence, $x\mathsf{ML}^\mathsf{F}$ can be used as an internal language for ML^F, with either semantics, and also for the many restrictions of ML^F that have been proposed, including HML. Hopefully, this will help the adoption of ML^F and maintain a powerful form of type inference in modern programming languages that must feature some form of first-class polymorphism.

Appendix A. Coq formalization

The Coq development is available electronically[11].

We have proved most of the meta-theoretical results of §2 and §3 using the Coq proof assistant (Coq development team, 2009). In order to deal with alpha-conversion issues—which often represent the most burdensome part of the formalization—we have used the *locally nameless* approach of Aydemir et al. (2008). In this setting, free variables are represented by names, while bound variables are De Bruijn indices. When going through a binder, a term must be *opened* by replacing the bound variable by a free variable. Of course, this variable must be fresh; this is ensured by a cofinite quantification, that allows all names but a given finite set, typically chosen to contain all the free variables of the local typing context.

Given the strong syntactical similarities between $x\mathsf{ML}^\mathsf{F}$ and $\mathsf{F}_{<:}$, notably the instance-bounded quantification, we have been able to reuse most of the definitions and results previously established for the examples of (Aydemir et al., 2008). Extending the formalism to add type instantiations was quite natural with a lot of cut-and-paste. We have however found it important to update the tactics[12] contained in the development so that they seamlessly handle the constructs we have added. This way, we have been able to reuse the very high level of automation they provide, which is quite striking in the initial development.

Up-to the use of the locally nameless formalism, our formalization is very faithful to the metatheory of §2. One small difference is that we did not define the operation $\tau\,\phi$ as a function, but as a relation. (See below for a justification.) Also, as it is painful to define reduction relations using evaluation contexts, we have inlined rule Context for each context. Finally, characterizing subrelations is also technically heavy, so we have not attempted to formally prove results about call-by-value and call-by-name, but only for $\longrightarrow$.

Unfortunately, we have also encountered some difficulties. In particular, defining the operation $a\{!\alpha \leftarrow \phi; !\alpha\}$ proved very complicated. To understand why, let us recall rule ι-Inside:

$$(\Lambda(\alpha \geqslant \tau)\ a)\ (\forall\,(\geqslant \phi)) \longrightarrow \Lambda(\alpha \geqslant \tau\ \phi)\ a\{!\alpha \leftarrow \phi; !\alpha\}$$

The problem lies in the fact that the instantiation $(\phi; !\alpha)$ is not closed in the locally nameless sense when it is substituted instead of $!\alpha$. That is, the variable α is not free, but bound in front of $a\{!\alpha \leftarrow \phi; !\alpha\}$. Since bound variables are De Bruijn indices, it is impossible to define the entire operation as a simple recursive operation on a. Instead,

[11] At the url `http://www.yakobowski.org/publis/2010/xmlf-coq/`.

[12] Coq proofs are done using a set of commands, called tactics, which describe in a very high-level way how to build proof terms. The locally nameless examples define some very specialized tactics, that handle *e.g.* the computation of the set of variables against which a variable must be fresh.

we need *e.g.* to shift $(\phi; !\alpha)$ when crossing a binder. However, this is unsatisfactory, as it requires a considerable amount of new metatheory related to shifting (which the locally nameless approach had been introduced to avoid!). We instead chose to temporarily close ϕ when doing the substitution, by replacing the bound variable α by a fresh free one. Still (and unsurprisingly), this was not sufficient, as inside the proofs the variable was not "fresh enough". We thus had to prove that using any fresh free variable, not just the first available one, was equivalent. Those renaming lemmas were quite tedious to prove.

Notice that the exact same problem theoretically occurs when defining the operation $\tau\,\phi$, for the rule $(\forall\,(\alpha \geqslant \tau)\ \tau')(\forall\,(\geqslant \alpha)\phi) = \forall\,(\alpha \geqslant \tau)\ (\tau'\,\phi)$. In this case, we did not introduce tedious renaming lemmas, but simply defined $\tau\,\phi$ as a relation, instead of as a function.

We tried using the the same solution for $a\{!\alpha \leftarrow \phi; !\alpha\}$, which solved some problems related to bound *v.s.* free variables. However, such a solution is only partial. Indeed, when proving progress, we need to give the result term to which a source term reduces to. For rule ι-Inside, we have to show that both $\tau\,\phi$ and the term $a\{!\alpha \leftarrow \phi; !\alpha\}$ exist. For $\tau\,\phi$, this is easily deduced from the typability of the original term, which requires $\Gamma \vdash \phi : \tau \leq \tau'$ to hold for some Γ. For $a\{!\alpha \leftarrow \phi; !\alpha\}$, this is unfortunately essentially as hard as defining the constructive version of the operation.

Appendix B. Proofs of §2.5.

Proof of Lemma 14

Let v be a value. If it is an abstraction or a type abstraction, the result is immediate. If v is a partially applied constant, and it is applied to less than its arity, it has either a type of the form $\forall\,(\alpha \geqslant \tau)\ \tau'$, or $\tau \to \tau'$. If it is a fully applied constructor, it cannot have type $\bot$ by hypothesis. ■

Proof of Theorem 15

The proof is quite standard and proceed by cases on a. Only the first case is original, but still proceeds without difficulties:

- if a is $a'\ \phi$, by inversion of typing a' is typable in the empty environment. If a' is not a value, it can be further reduced by Context, and so can a. Otherwise, we proceed by cases on ϕ:
  - if ϕ is $\mathbb{1}$, $\invamp$ or $\phi_1; \phi_2$, a can be reduced by rules ι-Id, ι-Intro or ι-Seq
  - the case $\phi = !\alpha$ is impossible in the empty environment;
  - the case $\phi = @\tau$ is also impossible, as a' is a value which cannot have type $\bot$ by Lemma 14.
  - in the three last cases, a' must have type $\forall\,(\alpha \geqslant \tau)\ \tau'$ for some τ and τ'. Since it is a value, by inversion of typing it is either a type abstraction of the form $\Lambda(\alpha \geqslant \tau)\ a''$ (and a can be reduced by ι-Inside, Under or ι-Elim), or it is a partially applied constants, and a is a value.
- if a is $a_1\ a_2$: by inversion of typing, a_1 and a_2 are typable in the empty environment, and a_1 has type $\tau \to \tau'$ for some τ and τ'. If a_1 or a_2 are not values, they can be further reduced, and a can be further reduced by Context. Otherwise, since a_1 is a value, of type $\tau \to \tau'$, we proceed by inversion of typing:
  - it a_1 is of the form $\lambda(x : \tau)\ a_1'$, a can be reduced by (β).
  - if a_1 is a partially applied primitive, either a is a fully applied primitive and it can be reduced by the appropriate δ rule, or a is a value.
  - if a_1 is a partially applied constructor: by hypothesis on the typing of constructors, a_1 is of the form $C\ \theta_1 \dots \theta_k\ v_1 \dots v_n$ with $n < |C|$ (as a full application would not have an arrow type). Then a is a value.
- if a is $\mathsf{let}\ x = a_2\ \mathsf{in}\ a_1$, by inversion of typing a_2 is typable in the empty environment. If it is not a value, by induction hypothesis it can be reduced. Hence, a can be reduced by rule Context. Otherwise, a can be reduced by rule (β_{let}).
- variables are not typable in the empty environment;
- constants, abstractions and type abstractions are values;

■

Proof of Theorem 18

By cases on a. The cases for variables, constants, abstractions, type abstractions and type applications are the same as for call-by-value.

- If a is $a_1\ a_2$: by inversion of typing, a_1 and a_2 are typable in the empty environment, and a_1 has type $\tau \to \tau'$ for some τ and τ'. If a_1 is not a value, by induction hypothesis it can be reduced, and so can a by rule Context. Otherwise, by inversion of typing and since a_1 is a value, it is either of the form $\lambda(x : \tau)\ a_1'$ (in which case a can be β-reduced), or a partially applied constant, and the reasoning is the same as for call-by-value.
- If a is $\mathsf{let}\ x = a_2\ \mathsf{in}\ a_1$, it can be reduced by rule (β_{let}).

■

Appendix C. A restriction of $x\mathsf{ML}^\mathsf{F}$ without variable bounds

A restriction of $x\mathsf{ML}^\mathsf{F}$ without variable bounds that is closed under reduction and in close correspondence with $e\mathsf{ML}^\mathsf{F}$ can still be defined a posteriori, by constraining the formation of terms.

The first idea to avoid variable bounds is to restrict the syntax of types and expressions as follows:

$$\begin{array}{lll} \rho ::= & \tau \to \tau \mid \forall\,(\alpha \geqslant \rho)\ \rho \mid \bot & \text{Constructed Types} \\ \tau ::= & \alpha \mid \rho \mid \forall\,(\alpha \geqslant \rho)\ \tau & \text{Types} \\ a ::= & \ldots \mid \Lambda(\alpha \geqslant \rho)\ a & \text{Terms} \\ \Gamma ::= & \varnothing \mid \Gamma, \alpha \geqslant \rho \mid \Gamma, x : \tau & \text{Environments} \end{array}$$

The typing rule for type abstraction can be restricted accordingly, replacing τ by ρ in bounds:

$$\textsc{TAbs}\quad \frac{\Gamma, \alpha \geqslant \rho \vdash a : \tau \qquad \alpha \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash \Lambda(\alpha \geqslant \rho)\ a : \forall\,(\alpha \geqslant \rho)\ \tau}$$

There is a slight difficulty however, because new variable bounds could be created during reduction by rule ι-Inside, turning a bound ρ into $\rho\ \phi$, which might be a variable. Indeed, assume $\alpha \geqslant \rho \vdash \phi : \rho' \leq \alpha$ (ϕ could be $@\alpha$ if ρ' is $\bot$ or of the form $\phi'; !\alpha$ with $\alpha \geqslant \rho \vdash \phi' : \rho' \leq \rho$) and consider the reduction sequence:

$$\begin{array}{lll} & \Lambda(\alpha \geqslant \rho)\ (\Lambda(\beta \geqslant \rho')\ a)\ (\forall\,(\geqslant \phi); \&) & (\mathbf{1}) \\ \longrightarrow & \Lambda(\alpha \geqslant \rho)\ (\Lambda(\beta \geqslant \alpha)\ a\{!\beta \leftarrow \phi; !\beta\})\ \& & (\mathbf{2}) \\ \longrightarrow & \Lambda(\alpha \geqslant \rho)\ a\{!\beta \leftarrow \phi; \mathbb{1}\}\{\beta \leftarrow \alpha\} & (\mathbf{3}) \end{array}$$

The term (1) is well-formed. However, after one reduction step the bound of β becomes a variable α and (2) is ill-formed. To prevent this from happening, we may restrict uses of ϕ inside bounds, replacing Rule ι-Inside by the following variant:

$$\textsc{Inst-Inside}\quad \frac{\Gamma \vdash \phi : \rho_1 \leq \rho_2}{\Gamma \vdash \forall\,(\geqslant \phi) : \forall\,(\alpha \geqslant \rho_1)\ \tau \leq \forall\,(\alpha \geqslant \rho_2)\ \tau}$$

As expected, this rejects the source term (1) as ill-typed. Unfortunately, this is too restrictive. For instance, it would also reject the application of a polymorphic function. When ρ and ρ' are $\bot$ and ϕ is $@\alpha$, (1) is a term of System F, which we must keep!

Notice that the ill-formed term (2) can be further reduced to the term (3), which is well-formed. This suggests another solution to recover type application: making $\forall\,(\geqslant \phi); \&$ a primitive instance operation, say $\$\phi$, and the above reduction sequence atomic, so that one does not see the intermediate ill-formed step.

In summary, $x\mathsf{ML}^\mathsf{F}_\flat$ is defined as follows: first we extend type instantiations with primitive type applications:

$$\phi ::= \ldots \mid \$\phi$$

$$\textsc{Inst-App}\quad \frac{\Gamma \vdash \phi : \rho \leq \tau}{\Gamma \vdash \$\phi : \forall\,(\alpha \geqslant \rho)\ \tau_0 \leq \tau_0\{\alpha \leftarrow \tau\}} \qquad \textsc{Inst-Bot}\quad \frac{}{\Gamma \vdash @\rho : \bot \leq \rho}$$

$$\textsc{Inst-Under}\quad \frac{\Gamma, \alpha \geqslant \rho \vdash \phi : \tau_1 \leq \tau_2}{\Gamma \vdash \forall\,(\alpha \geqslant)\ \phi : \forall\,(\alpha \geqslant \rho)\ \tau_1 \leq \forall\,(\alpha \geqslant \rho)\ \tau_2}$$

$$\textsc{Inst-Abstr}\quad \frac{\alpha \geqslant \rho \in \Gamma}{\Gamma \vdash !\alpha : \rho \leq \alpha} \qquad \textsc{Inst-Inside}\quad \frac{\Gamma \vdash \phi : \rho_1 \leq \rho_2}{\Gamma \vdash \forall\,(\geqslant \phi) : \forall\,(\alpha \geqslant \rho_1)\ \tau \leq \forall\,(\alpha \geqslant \rho_2)\ \tau}$$

$$\textsc{Inst-Elim}\quad \frac{}{\Gamma \vdash \& : \forall\,(\alpha \geqslant \rho)\ \tau' \leq \tau'\{\alpha \leftarrow \rho\}}$$

Figure C.19: Type instance for $x\mathsf{ML}^\mathsf{F}_\flat$

Accordingly, we add the reduction rule

$$(\Lambda(\alpha \geqslant \rho)\ a)\ (\$\phi) \longrightarrow a\{!\alpha \leftarrow \phi; \mathbb{1}\}\{\alpha \leftarrow \rho\ \phi\} \quad (\iota\text{-Type})$$

and the following case in the recursive definition of type instance:

$$(\forall\,(\alpha \geqslant \rho)\ \tau)\ (\$\phi)\ =\ \tau\{\alpha \leftarrow \rho\ \phi\}$$

so that $\$\phi$ behaves as its expanded form $(\forall\,(\geqslant \phi); \&)$. We then restrict the syntax of types and terms as described above, and type instantiation rules as described on Figure C.19 (Rules Inst-Intro, Inst-Comp, and Inst-Id are omitted as they are left unchanged).

Notice that the intermediate language after the extensions and before the restrictions, say $x\mathsf{ML}^\mathsf{F}_\sharp$, is equivalent to $x\mathsf{ML}^\mathsf{F}$: both typing and reduction rules of $\$\phi$ are derived; subject reduction hence holds in $x\mathsf{ML}^\mathsf{F}_\sharp$.

We show that reduction of $x\mathsf{ML}^\mathsf{F}_\sharp$ is closed in the $x\mathsf{ML}^\mathsf{F}_\flat$ subset by revisiting the proof of subject reduction for $x\mathsf{ML}^\mathsf{F}$, and checking in each case that the typing derivation rebuilt after reduction is well-formed in $x\mathsf{ML}^\mathsf{F}_\flat$, having ρ terms instead of general τ terms wherever required by the syntax and the typing rules of $x\mathsf{ML}^\mathsf{F}_\flat$.

Finally, the target of the translation of $e\mathsf{ML}^\mathsf{F}$ into $x\mathsf{ML}^\mathsf{F}$, described in §3, lies in $x\mathsf{ML}^\mathsf{F}_\flat$. In particular, bounds of variables are ρ-types $\mathcal{R}(\cdot)$. Moreover, the translation of instantiation witnesses described in Figure 17 only applies $@(\cdot)$ to ρ-types $\mathcal{R}(\cdot)$. Uses of $!\alpha$ appear either in expressions $\forall\,(\geqslant !\alpha); \&$, which can be replaced by $\$(!\alpha)$, or not under $\forall\,(\geqslant \cdot)$.

References

D. Le Botlan, D. Rémy, MLF: Raising ML to the power of System-F, in: Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP'03), ACM Press, 27–38, URL http://doi.acm.org/10.1145/944705.944709, 2003.

D. Le Botlan, D. Rémy, Recasting MLF, Information and Computation 207 (6) (2009) 726–785, ISSN 0890-5401, URL http://dx.doi.org/10.1016/j.ic.2008.12.006.

D. Rémy, B. Yakobowski, From ML to MLF: Graphic type constraints with efficient type inference, in: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08), ACM Press, 63–74, URL http://doi.acm.org/10.1145/1411203.1411216, 2008.

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

M. P. Jones, A theory of qualified types, Science of Computer Programming 22 (3) (1994) 231–256, ISSN 0167-6423, URL http://dx.doi.org/10.1016/0167-6423(94)00005-0.

G. Manzonetto, P. Tranquilli, Harnessing MLF with the Power of System F, in: Mathematical Foundations of Computer Science, vol. 6281 of *LNCS*, 525–536, URL http://perso.ens-lyon.fr/paolo.tranquilli/content/docs/snmlf.pdf, 2010.

B. Yakobowski, Graphical types and constraints: second-order polymorphism and inference, Ph.D. thesis, University of Paris 7, URL http://www.yakobowski.org/phd-dissertation.html, 2008.

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

D. Rémy, B. Yakobowski, A graphical presentation of MLF types with a linear-time unification algorithm, in: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI' 07), ACM Press, 27–38, URL http://doi.acm.org/10.1145/1190315.1190321, 2007.

G. Scherer, Extending MLF with Higher-Order Types, Master's thesis, École Normale Supérieure d'Ulm, URL http://gallium.inria.fr/~remy/mlf/scherer@master2010:mlfomega.pdf, 2010a.

F. Pottier, D. Rémy, The Essence of ML Type Inference, in: B. C. Pierce (Ed.), Advanced Topics in Types and Programming Languages, chap. 10, MIT Press, 389–489, URL http://cristal.inria.fr/attapl/, 2005.

G. Scherer, Prototype implementation of MLF, URL http://gallium.inria.fr/~remy/mlf/mlf-omega/, 2010b.

D. Leijen, Flexible types: robust type inference for first-class polymorphism, Tech. Rep. MSR-TR-2008-55, Microsoft Research, URL ftp://ftp.research.microsoft.com/pub/tr/TR-2008-55.pdf, 2008.

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

J. Cretin, D. Rémy, On the Power of Coercions Abstraction, in: ACM Symposium on Principles of Programming Languages (POPL), Philadephia, Pennsylvania, URL http://cristal.inria.fr/~remy/coercions/, 2012.

D. Leijen, A. Löh, Qualified types for MLF, in: Proceedings of the 10th International Conference on Functional Programming (ICFP '05), ACM Press, 144–155, URL http://doi.acm.org/10.1145/1090189.1086385, 2005.

D. Leijen, A Type Directed Translation of MLF to System F, in: Proceedings of the 12th International Conference on Functional Programming (ICFP '07), ACM Press, 111–122, URL http://doi.acm.org/10.1145/1291151.1291169, 2007.

K. Crary, Typed compilation of inclusive subtyping, in: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP '00), ACM Press, ISBN 1-58113-202-6, 68–81, URL http://doi.acm.org/10.1145/351240.351247, 2000.

M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, K. Donnelly, System F with type equality coercions, in: Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI' 07), ACM Press, 53–66, URL http://doi.acm.org/10.1145/1190315.1190324, 2007.

S. Weirich, D. Vytiniotis, S. Peyton Jones, S. Zdancewic, Generative type abstraction and type-level computation, in: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '11, ACM, New York, NY, USA, ISBN 978-1-4503-0490-0, 227–240, URL http://doi.acm.org/10.1145/1926385.1926411, 2011.

P. Herms, Partial Type Inference with Higher-Order Types, Master's thesis, University of Pisa and INRIA, URL http://pauillac.inria.fr/~remy/mlf/Herms@master2009:mlf-omega.pdf, 2009.

Coq development team, The Coq Proof Assistant Reference Manual, version 8.2, URL http://coq.inria.fr/refman/, 2009.

B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, S. Weirich, Engineering formal metatheory, in: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles Of Programming Languages (POPL '08), ACM Press, ISBN 978-1-59593-689-9, 3–15, URL http://doi.acm.org/10.1145/1328438.1328443, 2008.