Harnessing ML^{F} with the Power of System F

Giulio Manzonetto[1⋆] and Paolo Tranquilli[2⋆⋆]

[1] Laboratoire LIPN, CNRS UMR 7030
Université Paris Nord, France
giulio.manzonetto@lipn.univ-paris13.fr

[2] LIP, CNRS UMR 5668, INRIA,
ENS de Lyon, Université Claude Bernard Lyon 1, France
paolo.tranquilli@ens-lyon.fr

Abstract. We provide a strong normalization result for ML^{F}, a type system generalizing ML with first-class polymorphism as in system F. The proof is achieved by translating ML^{F} into a calculus of coercions, and showing that this calculus is just a decorated version of system F.
Keywords: ML^{F}, strong normalization, coercions, polymorphic types.

1 Introduction

One of the most efficient techniques for assuring that a program "behaves well" is *static type-checking*: types are assigned to every subexpression of a program, so that consistency of such an assignment (checked at compile time) implies well-behavedness. Such assignment may be *explicit*, i.e. requiring the programmer to annotate the types at key points in the program (e.g. variables), as in C or Java. Otherwise we can free the programmer of the hassle and leave to an automatic type reconstructor, part of the compiler, the boring part of scattering the code with types. One of the most prominent examples of this approach is ML [1] and its dialects, a functional programming language, as such based on λ-calculus.

In this context *type polymorphism* allows greater flexibility, making it possible to reuse code that works with elements of different types. For example an identity function will have type $\alpha \to \alpha$ for any α, so one can give it the type $\forall(\alpha)(\alpha \to \alpha)$. However full polymorphism (like in system F [2]) leads to undecidable type systems: no automatic reconstructor would be available. For this reason ML has the so called second-class polymorphism (i.e. available only for named variables), more restricted but allowing a type inference procedure. Unfortunately, the programmer is also *forced* to use second-class polymorphism only. One could wish for a more flexible approach, where one would write just enough type annotations to let the compiler's type reconstructor do the job, while still being able to employ first-class polymorphism, if desired.

ML^{F} [3] answers this call by providing a partial type annotation mechanism with an automatic type reconstructor. This extension allows to write system F

⋆ Supported by Digiteo/Île-de-France project COLLODI (2009-28HD).
⋆⋆ Supported by ANR project COMPLICE (ANR-08-BLANC-0211-01).

programs, which is not possible in general in ML, while remaining conservative: ML programs still type-check without needing any annotation. An important feature are principal type schemata, lacking in system F, which are obtained by employing a downward bounded quantification $\forall(\alpha \geq \sigma)\tau$, called a *flexible* quantifier. Such a type intuitively denotes that τ may be instantiated to any $\tau\{\sigma'/\alpha\}$, *provided that* σ' *is an instantiation of* σ. Usual quantification is recovered by allowing $\bot$ (morally equivalent to $\forall\alpha.\alpha$) as bound. ML^F also uses a *rigid* quantifier $\forall(\alpha = \sigma)\tau$, fundamental for type inference but not for the semantics[3].

One of the properties of well-behavedness that a type system can assure is *strong normalization* (SN), that is the termination of all typable programs whatever execution strategy is used. For example, system F is strongly normalizing. As already pointed out, system F is contained in ML^F; it is not yet known, but it is conjectured [3], that the inclusion is strict. This makes the question of SN of ML^F a non-trivial one, to which we answer positively in this paper. The result is proved via a suitable simulation in system F, with additional decorations dealing with the complex type instantiations possible in ML^F.

Our starting point is xML^F [4], the Church version of ML^F, briefly presented in section 2. In xML^F type inference and the rigid quantifier $\forall(\alpha = \sigma)\tau$ are abandoned, with the aim of providing an internal language to which a compiler might map the surface language briefly presented above (which in fact is denoted more precisely by eML^F[4]). Compared to Church-style system F, the type reduction $\to_\iota$ of xML^F is more complex, and may *a priori* cause unexpected glitches: it could cause non-termination, or block the reduction of a β-redex. To prove that none of this happens, we use as target language of our translation a decoration of system F, the *coercion calculus* $\mathsf{F_c}$, which has its own interest. Indeed, xML^F has syntactic entities (the *instantiations* ϕ) testifying an instance relation between types, and it is natural to regard them as coercions. The delicate point is that some of these instantiations (the "abstractions" $!\alpha$) behave in fact as variables, abstracted when introducing a bounded quantifier: in a way, $\forall(\alpha \geq \sigma)\tau$ expects a coercion from σ to α, whatever the choice for α may be.

A question naturally arising is: what does it mean to be a coercion in this context, where such operations of coercion abstraction and substitution are available? Our answer, which works for xML^F, is in the form of a type system ($\mathsf{F_c}$, Figure 2). In section 3 we will show the good properties enjoyed by $\mathsf{F_c}$: it is a decoration of system F, so it is SN; moreover it has a *coercion erasure* which ideally recovers the actual semantics of a term, and establishes a *weak bisimulation* with system F, where coercion reductions $\to_\mathsf{c}$ take the role of silent actions, while β-reduction $\to_\beta$ remains the observable one.

The generality of coercion calculus allows then to lift these results to xML^F via the above mentioned translation (section 4). The main idea of the translation is the same as the one shown for eML^F in [5], where however no dynamic property

[3] Indeed $\forall(\alpha = \sigma)\tau$ can be regarded as being $\tau\{\sigma/\alpha\}$.

[4] There is also a completely annotation-free version, iML^F, clearly at the cost of loosing type inference. For details on the different versions of ML^F, the reader may be referred to `http://gallium.inria.fr/~remy/mlf/`.

was studied. Here we finally produce a proof of SN for all versions of $\mathsf{ML^F}$. Moreover the bisimulation result for $\mathsf{xML^F}$ establishes once and for all that it can be used as an internal language for $\mathsf{eML^F}$, as the additional type structure cannot block reductions of the intended program.

Notations. Given reductions $\to_1$ and $\to_2$, we write $\to_1\to_2$ (resp. $\to_{12}$) for their concatenation (resp. their union). Moreover $\leftarrow$, $\xrightarrow{+}$, $\xrightarrow{=}$ and $\xrightarrow{*}$ denote the transpose, the transitive, the reflexive and the transitive-reflexive closures of $\to$ respectively. In confluence diagrams, solid arrows denote reductions one starts with, while dashed arrows are the entailed ones.

2 A Short Introduction to $\mathsf{ML^F}$and its Variants

Currently, $\mathsf{ML^F}$ comes in a Curry-style version $\mathsf{iML^F}$, where no type information is needed, and a type-inference version $\mathsf{eML^F}$ requiring partial type information. However, $\mathsf{eML^F}$ is not completely in Church-style, since a large amount of type information is still inferred. A truly Church-style version of $\mathsf{ML^F}$, called $\mathsf{xML^F}$, has been recently introduced in [4] and will be our main object of study in this paper. However, we will draw conclusions for $\mathsf{iML^F}$ and $\mathsf{eML^F}$ too.

All the syntactic definitions of $\mathsf{xML^F}$ can be found in Figure 1. Types include: usual variable and arrow types; a type $\bot$ corresponding to system F's type $\forall\alpha.\alpha$; the *flexible quantification* $\forall(\alpha \geq \sigma)\tau$ generalizing $\forall\alpha.\tau$ of system F. Intuitively, $\forall(\alpha \geq \sigma)\tau$ restricts the variable α to range just over instances of σ. The variable α is bound in τ but not in σ. The instantiation ϕ maps a type σ to a type τ which is an instance of σ. Thus ϕ can be seen as a 'witness' of the instance relation holding between σ and τ. In $\forall(\alpha \geq)\phi$, α is bounded in ϕ.

Environments Γ are finite maps assigning types (resp. bounds) to term (resp. type) variables. We write: $\mathrm{dom}(\Gamma)$ for the set of all term and type variables that are bound by Γ; $\mathrm{ftv}(\tau)$ for the set of type variables appearing free in τ. Environments of shape $\Gamma, \alpha \geq \tau, \Gamma'$ or $\Gamma, x : \tau, \Gamma'$ are *well-formed* if $\mathrm{ftv}(\tau) \subseteq \mathrm{dom}(\Gamma)$. All environments in this paper are supposed to be well-formed.

Reduction rules are divided into $\to_\beta$ (regular β-reductions) and $\to_\iota$, reducing instantiations. We recall (from [4, Sec. 2.1]) that both $\to_\beta$ and $\to_\iota$ enjoy subject reduction. One of the ι-steps uses the definition of type instantiation $\tau\phi$, giving the unique type such that $\Gamma \vdash \phi : \tau \leq \tau\phi$, if ϕ type-checks.

Convention 1. The `let` construct is added mainly to accommodate $\mathsf{eML^F}$'s type reconstructor. Thus, in the whole paper we suppose that in all $\mathsf{xML^F}$ terms every $\mathtt{let}\, x = a \,\mathtt{in}\, b$ has been replaced by $(\lambda(x : \sigma)b)a$, with σ the correct type of a.

The *type erasure* $\lceil a \rceil$ of an $\mathsf{xML^F}$ (or $\mathsf{eML^F}$) term a is straightforwardly defined by erasing all type and instantiation annotations, mapping a to an ordinary λ-term. From [4, Lemma 7, Theorem 6 and §4.2] we know the following.

Theorem 2. *For every $\mathsf{iML^F}$ or $\mathsf{eML^F}$ term a, there is an $\mathsf{xML^F}$ term $[\![a]\!]$ such that $\lceil [\![a]\!] \rceil = \lceil a \rceil$*[5]*.*

[5] We only need to apply type erasure on the right for partially annotated $\mathsf{eML^F}$ terms.

Syntactic definitions

$$
\begin{array}{llr}
\alpha, \beta, \dots & & \text{(type variables)} \\
\sigma, \tau & ::= \alpha \mid \sigma \to \tau \mid \bot \mid \forall(\alpha \geq \sigma)\tau & \text{(types)} \\
\phi, \psi & ::= \tau \mid \phi;\psi \mid \mathbf{1} \mid \& \mid \parr & \\
 & \quad \mid\ !\alpha \mid \forall(\geq \phi) \mid \forall(\alpha \geq)\phi & \text{(instantiations)} \\
\Gamma & ::= \emptyset \mid \Gamma, \alpha \geq \tau \mid \Gamma, x : \tau & \text{(environments)}
\end{array}
\qquad
\begin{array}{llr}
x, y, z, \dots & & \text{(variables)} \\
a, b, c & ::= x \mid \lambda(x:\tau)a \mid ab & \\
 & \quad \mid \Lambda(\alpha \geq \tau)a \mid a\phi & \text{(terms)} \\
 & \quad \mid \mathtt{let}\, x = a\, \mathtt{in}\, b & \\
A, B & ::= a \mid \phi & \text{(expressions)}
\end{array}
$$

Instantiation rules

$$
\frac{}{\Gamma \vdash \tau : \bot \leq \tau}\,\textsc{IBot}
\qquad
\frac{\Gamma, \alpha \geq \tau \vdash \phi : \tau_1 \leq \tau_2}{\Gamma \vdash \forall(\alpha \geq)\phi : \forall(\alpha \geq \tau)\tau_1 \leq \forall(\alpha \geq \tau)\tau_2}\,\textsc{IUnder}
$$

$$
\frac{\alpha \geq \tau \in \Gamma}{\Gamma \vdash !\alpha : \tau \leq \alpha}\,\textsc{IAbs}
\qquad
\frac{\Gamma \vdash \phi : \tau_1 \leq \tau_2}{\Gamma \vdash \forall(\geq \phi) : \forall(\alpha \geq \tau_1)\tau \leq \forall(\alpha \geq \tau_2)\tau}\,\textsc{IInside}
$$

$$
\frac{\alpha \notin \mathrm{ftv}(\tau)}{\Gamma \vdash \parr : \tau \leq \forall(\alpha \geq \bot)\tau}\,\textsc{IIntro}
\qquad
\frac{}{\Gamma \vdash \& : \forall(\alpha \geq \sigma)\tau \leq \sigma\,\{\tau/\alpha\}}\,\textsc{IElim}
$$

$$
\frac{\Gamma \vdash \phi : \tau_1 \leq \tau_2 \qquad \Gamma \vdash \psi : \tau_2 \leq \tau_3}{\Gamma \vdash \phi;\psi : \tau_1 \leq \tau_3}\,\textsc{IComp}
\qquad
\frac{}{\Gamma \vdash \mathbf{1} : \tau \leq \tau}\,\textsc{IId}
$$

Typing rules

$$
\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\,\textsc{Var}
\qquad
\frac{\Gamma \vdash a : \tau \qquad \Gamma, x : \tau \vdash b : \sigma}{\Gamma \vdash \mathtt{let}\, x = a\, \mathtt{in}\, b : \sigma}\,\textsc{Let}
$$

$$
\frac{\Gamma, x : \tau \vdash a : \sigma}{\Gamma \vdash \lambda(x:\tau)a : \tau \to \sigma}\,\textsc{Abs}
\qquad
\frac{\Gamma \vdash a : \sigma \to \tau \qquad \Gamma \vdash b : \sigma}{\Gamma \vdash ab : \tau}\,\textsc{App}
$$

$$
\frac{\Gamma, \alpha \geq \sigma \vdash a : \tau \qquad \alpha \notin \mathrm{ftv}(\Gamma)}{\Gamma \vdash \Lambda(\alpha \geq \sigma)a : \forall(\alpha \geq \sigma)\tau}\,\textsc{TAbs}
\qquad
\frac{\Gamma \vdash a : \tau \qquad \Gamma \vdash \phi : \tau \leq \sigma}{\Gamma \vdash a\phi : \sigma}\,\textsc{TApp}
$$

Type instantiation

$$
\begin{array}{llll}
\tau(!\alpha) := \alpha, & \bot\tau := \tau, & \tau\mathbf{1} := \tau, & \tau(\phi;\psi) := (\tau\phi)\psi, \\
\tau\parr := \forall(\alpha \geq \bot)\tau, & \alpha \notin \mathrm{ftv}(\tau), & & (\forall(\alpha \geq \sigma)\tau)\& := \tau\,\{\sigma/\alpha\}, \\
\multicolumn{2}{l}{(\forall(\alpha \geq \sigma)\tau)(\forall(\geq \phi)) := \forall(\alpha \geq \sigma\phi)\tau,} & \multicolumn{2}{l}{(\forall(\alpha \geq \sigma)\tau)(\forall(\alpha \geq)\phi) := \forall(\alpha \geq \sigma)(\tau\phi).}
\end{array}
$$

Reduction rules

$$
\begin{array}{rlrl}
(\lambda(x:\tau)a)b & \to_\beta a\,\{x/b\} & a\parr & \to_\iota \Lambda(\alpha \geq \bot)a, \quad \alpha \notin \mathrm{ftv}(\tau) \\
\mathtt{let}\, x = b\, \mathtt{in}\, a & \to_\beta a\,\{x/b\} & (\Lambda(\alpha \geq \tau)a)\& & \to_\iota a\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\} \\
a\mathbf{1} & \to_\iota a & (\Lambda(\alpha \geq \tau)a)(\forall(\alpha \geq)\phi) & \to_\iota \Lambda(\alpha \geq \tau)(a\phi) \\
a(\phi;\psi) & \to_\iota (a\phi)\psi & (\Lambda(\alpha \geq \tau)a)(\forall(\geq \phi)) & \to_\iota \Lambda(\alpha \geq \tau\phi)a\,\{\phi;!\alpha/!\alpha\}
\end{array}
$$

Fig. 1: Syntactic definitions, typing and reduction rules of $\mathsf{xML^F}$.

3 The Coercion Calculus $\mathsf{F_c}$

In this section we will introduce the *coercion calculus* $\mathsf{F_c}$, which is (as shown in subsection 3.2) a decoration of system F accompanied by a type system. Before introducing the details, we point out that the version of $\mathsf{F_c}$ presented here is tailored down to suit $\mathsf{xML^F}$. As such, there are natural choices that have been intentionally left out or restrained. If $\mathsf{F_c}$ is to serve as a good metatheory of coercions, more liberal choices and constructs are needed, as discussed at page 12. The syntax, the type system and the reduction rules of $\mathsf{F_c}$[6] are presented in

Syntactic definitions

$\alpha, \beta, \dots$	(type variables)
$\sigma, \tau ::= \alpha \mid \sigma \to \tau \mid \kappa \to \tau \mid \forall\alpha.\tau$	(types)
$\kappa \quad ::= \sigma \multimap \tau$	(coercion types)
$\zeta \quad ::= \tau \mid \kappa$	(type expr.)
$x, y, z, \dots$	(variables)
$a, b ::= x \mid \lambda x.a \mid \underline{\lambda} x.a \mid \underline{\measuredangle} x.a \mid ab \mid a \triangleright b \mid a \triangleleft b$	(terms)
$u, v ::= \lambda x.a \mid \underline{\measuredangle} x.u \mid x \triangleright u$	(c-values)
$\Gamma ::= \emptyset \mid x : \tau, \Gamma \mid x : \sigma \multimap \alpha, \Gamma$	(regular env.)
$L ::= \emptyset \mid z : \tau$	(linear env.)
$\Gamma; L$	(environments)
$\Gamma; \vdash_{\mathtt{t}} a : \sigma$	(term judgements)
$\Gamma; \vdash_{\mathsf{c}} a : \sigma \multimap \tau$	(coercion judgements)
$\Gamma; z : \tau \vdash_{\ell} a : \sigma$	(linear judgements)

$\vdash_{\mathtt{xy}}$, $\mathtt{x}, \mathtt{y} \in \{\mathtt{t}, \mathsf{c}, \ell\}$ stands for $\vdash_{\mathtt{x}}$ or $\vdash_{\mathtt{y}}$.

Typing rules

$$\frac{\Gamma(y) = \zeta}{\Gamma; \vdash_{\mathtt{t}\ell} y : \zeta}\,\textsc{Ax} \qquad \frac{\Gamma, x : \tau; \vdash_{\mathtt{t}} a : \sigma}{\Gamma; \vdash_{\mathtt{t}} \lambda x.a : \tau \to \sigma}\,\textsc{Abs} \qquad \frac{\Gamma; \vdash_{\mathtt{t}} a : \sigma \to \tau \qquad \Gamma; \vdash_{\mathtt{t}} b : \sigma}{\Gamma; \vdash_{\mathtt{t}} ab : \tau}\,\textsc{App}$$

$$\frac{}{\Gamma; z : \tau \vdash_{\ell} z : \tau}\,\textsc{LAx} \qquad \frac{\Gamma; z : \tau \vdash_{\ell} a : \sigma}{\Gamma; \vdash_{\mathsf{c}} \underline{\lambda} z.a : \tau \multimap \sigma}\,\textsc{LAbs} \qquad \frac{\Gamma, x : \kappa; L \vdash_{\mathtt{t}\ell} a : \sigma}{\Gamma; L \vdash_{\mathtt{t}\ell} \underline{\measuredangle} x.a : \kappa \to \sigma}\,\textsc{CAbs}$$

$$\frac{\Gamma; \vdash_{\mathsf{c}} a : \sigma_1 \multimap \sigma_2 \qquad \Gamma; L \vdash_{\mathtt{t}\ell} b : \sigma_1}{\Gamma; L \vdash_{\mathtt{t}\ell} a \triangleright b : \sigma_2}\,\textsc{LApp} \qquad \frac{\Gamma; L \vdash_{\mathtt{t}\ell} a : \kappa \to \sigma \qquad \Gamma \vdash_{\mathsf{c}} b : \kappa}{\Gamma; L \vdash_{\mathtt{t}\ell} a \triangleleft b : \sigma}\,\textsc{CApp}$$

$$\frac{\Gamma; L \vdash_{\mathtt{t}\ell} a : \sigma \qquad \alpha \notin \mathrm{ftv}(\Gamma; L)}{\Gamma; L \vdash_{\mathtt{t}\ell} a : \forall\alpha.\sigma}\,\textsc{Gen} \qquad \frac{\Gamma; L \vdash_{\mathtt{t}\ell} a : \forall\alpha.\sigma}{\Gamma; L \vdash_{\mathtt{t}\ell} a : \sigma\{\tau'/\alpha\}}\,\textsc{Inst}$$

Reduction rules

$$(\lambda x.a)b \to_{\beta} a\{b/x\}, \qquad (\underline{\measuredangle} x.a) \triangleleft b \to_{\mathsf{c}} a\{b/x\}, \qquad (\underline{\lambda} x.a) \triangleright b \to_{\mathsf{c}} a\{b/x\},$$

$$(\underline{\measuredangle} x.u) \triangleleft b \to_{\mathsf{cv}} u\{b/x\}, \qquad (\underline{\lambda} x.a) \triangleright u \to_{\mathsf{cv}} a\{u/x\}, \qquad \text{if } u \text{ is a } \mathsf{c}\text{-value}.$$

Fig. 2: Syntactic definitions, typing and reduction rules of coercion calculus.

Figure 2. In this calculus the notion of 'coercion' is captured by suitable types.

Definition 3 (Coercion). *An* $\mathsf{F_c}$ *term* a *is a* coercion *if* $\Gamma; \vdash_{\mathsf{c}} a : \sigma \multimap \tau$*.*

The use of linear implication for the type of coercions is not casual. Indeed the type system can be seen as a fragment of DILL, the dual intuitionistic linear logic [6]. This captures an aspect of coercions: they consume their argument without erasing it (as they must preserve it) nor duplicate it (as there is no true computation, just a type recasting). *Environments* are of shape $\Gamma; L$, where Γ is a map from variables to type expressions, and L is the *linear* part of the environment, containing (contrary to DILL) *at most* one assignment. Notice the restriction to $\sigma \multimap \alpha$ for coercion variables, which might at first seem overtly restrictive. However, Theorem 21 relies on this restriction, though the preceding results do not. Alternative, more permissive restrictions preserving the bisimulation result are left for future work.

Typing judgments come in three sorts. However, the subscripts we use to distinguish them ($\vdash_{\mathtt{t}}$, $\vdash_{\mathsf{c}}$ and $\vdash_{\ell}$) are only for easy recognition, as the sort of the judgment can be recovered from the shape of the environment and the type.

[6] We present the coercion calculus in Curry-style, whereas arguably its usefulness outside of this work would rather be in Church-style (which is easy to define).

A note on DILL *and λ-calculus.* The language presented in [6] is the term calculus of the logical system, and as such has a constructor for every logical rule. Notably, that work provides no intuitionistic arrow, as the translation $A \to B \cong\, !A \multimap B$ is preferred. Employing DILL as a type system for ordinary λ-terms leads to a system (which we might call F_ℓ) using types rather than terms to strictly differentiate between linear and regular constructs. This system is known as *folklore*[7] but, as far as we know, it has never been studied in the literature. The absence of a thorough presentation of F_ℓ prevents us from deriving properties such as subject reduction (Proposition 8) more or less directly from a more general framework. We leave to further work the rather straightforward presentation of such a system together with a more general version of F_c, along the lines hinted at page 12.

Syntax. F_c terms are extensions of usual λ-terms with two abstractions $\underline{\lambda}$, $\underline{\measuredangle}$ and two applications $\triangleright$, $\triangleleft$. The *linear abstraction* $\underline{\lambda}$ (whose application is $\triangleright$) is used by coercions to ask for the regular term to coerce, so they cannot erase or duplicate it. The *coercion abstraction* $\underline{\measuredangle}$ (whose application is $\triangleleft$) can be used in regular or coercion terms to ask for a coercion, so it is not subject to particular restrictions. The applications $\triangleright$, $\triangleleft$ locate coercions within the terms without carrying the typing around: the triangle's side indicates where the coercion is.

Reductions. Reduction steps are divided into $\to_\beta$ (the actual computation) and $\to_\mathsf{c}$ (the coercion reduction). The reduction $\to_\mathsf{c}$ has a conditional subreduction $\to_\mathsf{cv}$ that fires c-redexes only when c-values are at the right of the $\triangleright$ or left of the $\triangleleft$. Intuitively, this reduction is what is strictly necessary to "unearth" a λ-abstraction. Its main role here is that it is general enough to have bisimulation (Theorem 21) and small enough to correspond to xML^F's ι-steps (Lemma 28). As usual, rules are closed by context.

3.1 Some Basic Properties of F_c

We start presenting some basic properties of the coercion calculus. The first statements restrain the shape and the behaviour of coercions.

Remark 4. A coercion a is necessarily either a variable or a coercion abstraction, as Ax and LAbs are the only rules having a coercion type in the conclusion.

proof in tech. app. ←

Lemma 5. *If $\Gamma; L \vdash_{\mathsf{c}\ell} a : \zeta$ then no subterm of a is of the form $\lambda x.b$ or bc. In particular a is β-normal.*

Lemma 6. *Let a be an F_c term. If $\Gamma; \vdash_\mathsf{c} a : \sigma \multimap \tau$, then a is* cv*-normal.*

Proof. Immediate by Lemma 5: there cannot be any subterm $\lambda x.a'$ of a, so in particular a does not contain any c-value. □

[7] As an example we might cite [7], where a fragment of F_ℓ is used to characterize polytime functions.

Following are basic properties of type systems. Note that though there are two substitution results (points (ii), (iii) below) to accommodate the two types of environment, no weakening property is available to add the linear assignment.

Lemma 7 (Weakening and substitution). *We have the following:*

→ proof in tech. app.

(i) $\Gamma; L \vdash_{\mathtt{tc}\ell} a : \zeta$ *and* $x \notin \mathrm{dom}(\Gamma; L)$ *entail* $\Gamma, x : \zeta'; L \vdash_{\mathtt{tc}\ell} a : \zeta$*;*
(ii) $\Gamma; \vdash_{\mathtt{tc}} a : \zeta'$ *and* $\Gamma, x : \zeta'; L \vdash_{\mathtt{tc}\ell} b : \zeta$ *entail* $\Gamma; L \vdash_{\mathtt{tc}\ell} b\,\{a/x\} : \zeta$*;*
(iii) $\Gamma; L \vdash_{\mathtt{t}\ell} a : \sigma$ *and* $\Gamma; x : \sigma \vdash_{\ell} b : \zeta$ *entail* $\Gamma; L \vdash_{\mathtt{t}\ell} b\,\{a/x\} : \zeta$*.*

Proposition 8 (Subject reduction). *If* $a \to_{\beta\mathsf{c}} b$ *and* $\Gamma; L \vdash_{\mathtt{t}\ell\mathtt{c}} a : \zeta$*, then* $\Gamma; L \vdash_{\mathtt{t}\ell\mathtt{c}} b : \zeta$*.*

→ proof in tech. app.

Proposition 9 (Confluence). *All of* $\to_\beta$*,* $\to_{\mathsf{c}}$*,* $\to_{\mathsf{cv}}$ *and* $\to_{\beta\mathsf{c}}$ *are confluent.*

Proof. The proof by Tait-Martin Löf's technique of parallel reductions does not pose particular issues. ☐

3.2 Coercion Calculus as a Decoration of System F

The following definition presents the coercion calculus as a simple decoration of usual Curry-style system F. The latter can be recovered by just collapsing the extraneous constructs $\multimap$, $\underline{\lambda}$, $\underline{\Lambda}$, $\triangleleft$ and $\triangleright$ to their regular counterpart. Notably this will lead to a strong normalization result.

Definition 10. *The* decoration erasure *is defined by:*

$$|\alpha| := \alpha, \quad |\zeta \to \tau| := |\zeta| \to |\tau|, \quad |\sigma \multimap \tau| := |\sigma| \to |\tau|,$$
$$|x| := x, \quad |\lambda x.a| = |\underline{\lambda} x.a| = |\underline{\Lambda} x.a| := \lambda x.|a|, \quad |a \triangleleft b| = |a \triangleright b| = |ab| := |a||b|,$$
$$|\Gamma|(y) := |\Gamma(y)| \quad \textit{for } y \in \mathrm{dom}(\Gamma), \qquad |\Gamma; z : \tau| := |\Gamma|, z : |\tau|.$$

The next lemma ensures that the decoration erasure preserves typability (with system F's typability denoted by $\vdash_{\mathsf{F}}$).

Lemma 11. *Let* a *be an* $\mathsf{F_c}$ *term. If* $\Gamma; L \vdash_{\mathtt{t}\ell} a : \zeta$ *then* $|\Gamma; L| \vdash_{\mathsf{F}} |a| : |\zeta|$*.*

Proof. It suffices to see that through $|\,.\,|$ all the new rules collapse to their regular counterpart: LAx becomes Ax, CAbs, LAbs become Abs, and CApp, LApp become App. In the latter cases the weakening lemma for $\vdash_{\mathsf{F}}$ may have to be applied to add the missing $z : |\tau|$ to one of the two branches. ☐

Lemma 12. *Given an* $\mathsf{F_c}$ *term* a *we have* $|a|\,\{|b|/x\} = |a\,\{b/x\}\,|$*. Moreover, if* $a \to_{\beta\mathsf{c}} b$ *then* $|a| \to |b|$*. The converse is also true if* a *is typable.*

Proof. The first two claims are immediate. The converse needs the typability hypothesis: take $|a| = (\lambda x.b_1')b_2'$, then there are b_i with $|b_i| = b_i'$ and a is one of nine combinations ($(\lambda x.b_1)b_2$, $(\underline{\lambda} x.b_1)b_2$, $(\lambda x.b_1) \triangleleft b_2$, etc.). However as a is typable only the three matching combinations are possible, giving rise to the three possible redexes in the coercion calculus. ☐

Corollary 13 (Termination). *The coercion calculus is strongly normalizing.*

Proof. Immediate by Lemmas 11 and 12, using the strong normalization of system F [2, Sec. 14.3]. ☐

3.3 Preservation of the Semantics

We will now turn to establishing why coercions $a : \tau \multimap \sigma$ can be truly called such. First, we need a way to extract the semantics of a term, i.e., a way to strip it of the structure one may have added to it in order to manage coercions.

Definition 14. *The* coercion erasure *is defined by*

$$\lfloor x \rfloor := x, \quad \lfloor \lambda x.a \rfloor := \lambda x.\lfloor a \rfloor, \quad \lfloor ab \rfloor := \lfloor a \rfloor \lfloor b \rfloor,$$
$$\lfloor \underline{\lambda} x.a \rfloor = \lfloor \underline{\Lambda} x.a \rfloor := \lfloor a \rfloor, \quad \lfloor a \triangleleft b \rfloor := \lfloor a \rfloor, \quad \lfloor a \triangleright b \rfloor := \lfloor b \rfloor.$$

proof in tech. app. ←

Lemma 15.

(i) If $\Gamma, x : \kappa; L \vdash_{\mathtt{t}\ell} a : \sigma$ then $x \notin \mathrm{fv}(\lfloor a \rfloor)$;
(ii) if $\Gamma; z : \tau \vdash_\ell a : \sigma$ then $\lfloor a \rfloor = z$.

Notice that property (i) above entails that $\lfloor\,.\,\rfloor$ is well-defined with respect to α-equivalence on regular, typed terms: given a term $\underline{\Lambda} x.a$ issued from a coercion abstraction, $\lfloor \underline{\Lambda} x.a \rfloor = \lfloor a \rfloor$ is independent from x. This is not the case for coercions, as for example $\lfloor \underline{\lambda} x.x \rfloor = x$.

As for property (ii), it greatly restricts the form of a coercion: if $a : \sigma \multimap \tau$ then it is either a variable or an abstraction $\underline{\lambda} x.a'$ (as already written in Remark 4), with $\lfloor a' \rfloor = x$. Apart when they are variables, coercions are essentially identities.

One may ask whether the erasure maps $\mathsf{F_c}$ to a larger set of terms than system F. We do not know yet, though we conjecture it is the case.

Conjecture 16. There is an $\mathsf{F_c}$ term a such that $\lfloor a \rfloor$ is not typable in system F.

A note on unrestricted coercion variables. If we dropped the condition on coercion variables, namely that they are typed $\sigma \multimap \alpha$ in the context, we would get even more than the above conjecture, but too much, as the coercion erasure would cover the whole of the untyped λ-calculus. It would suffice to use two coercion variables $y_{o \to o} : o \multimap (o \to o)$ and $y_o : (o \to o) \multimap o$ modelling the recursive type $o \to o \simeq o$. For example, we would have $a_\delta := y_o \triangleright (\lambda x.(y_{o\to o} \triangleright x)x) : o$ and $a_\Delta := (y_{o \to o} \triangleright a_\delta)a_\delta : o$, though $\lfloor a_\Delta \rfloor = (\lambda x.xx)(\lambda x.xx)$ is the renown divergent and untypable term.

Lemma 17. $\lfloor a\,\{b/x\} \rfloor = \lfloor a \rfloor\,\{\lfloor b \rfloor/x\}$.

Proof. Immediate induction. □

proof in tech. app. ←

Lemma 18. *If $\Gamma; x : \tau \vdash_\ell a : \sigma$ and $b \to_\beta c$, then $a\,\{b/x\} \to_\beta a\,\{c/x\}$.*

Proof (sketch). Essentially the proof is by linearity of x in a. Formally it is carried out by an easy induction on the derivation. □

The following will state some basic dynamic properties of coercion reductions. Intuitively we will prove that β-steps are actual steps of the semantics (point (ii)) and that c-steps preserves it in a strong sense: they are collapsed to the equality (point (iii)) and they preserve β-steps (point (i)).

proof in tech. app. ←

Proposition 19. *Suppose that a is an $\mathsf{F_c}$ term. Then:*

(i) if $b_1 \leftarrow_{\mathtt{c}} a \rightarrow_\beta b_2$ then there is c with $b_1 \rightarrow_\beta c \overset{*}{\leftarrow}_{\mathtt{c}} b_2$;
(ii) if $a \rightarrow_\beta b$ then $\lfloor a \rfloor \rightarrow \lfloor b \rfloor$;
(iii) if $a \rightarrow_{\mathtt{c}} b$ then $\lfloor a \rfloor = \lfloor b \rfloor$.

$$\begin{tikzcd} a \arrow[r, "\beta"] \arrow[d, "\mathtt{c}"'] & b_2 \arrow[d, dashed, "\mathtt{c}*"] \\ b_1 \arrow[r, dashed, "\beta"] & c \end{tikzcd}$$

In order to truly see coercions as additional information that is not strictly needed for reduction, one may ask that some converse of property (ii) should also hold. Here the condition on coercion variables ($x : \sigma \multimap \alpha$) starts to play a role[8]. Indeed in general this is not the case: take $a = \underline{\lambda} y.(y \triangleright I)I$ with $I = \lambda x.x$, that would be typable with $;\vdash a : (\sigma_{\mathrm{id}} \multimap \sigma_{\mathrm{id}}) \rightarrow \sigma_{\mathrm{id}}$ (where $\sigma_{\mathrm{id}} := \forall\alpha.(\alpha \rightarrow \alpha)$). Its coercion erasure is typable but it has a redex that is blocked by a coercion variable.

With the condition on coercion variables in place we are ready to prove a complete correspondence between the β-reductions of the coerced terms and the ones of their coercion erasure. In fact Theorem 21 states that $a \mapsto \lfloor a \rfloor$ is a weak bisimulation for $\rightarrow_\beta$, taking $\rightarrow_{\mathtt{cv}}$ as the silent actions on the side of coercion calculus. The proof uses the following lemma.

Lemma 20. *Every typable $\mathtt{cv}$-normal term a such that $\lfloor a \rfloor = \lambda x.b$ is a $\mathtt{c}$-value. In particular if a has an arrow type then $a = \lambda x.c$ with $\lfloor c \rfloor = b$.* → proof in tech. app.

Theorem 21 (Bisimulation of $\lfloor . \rfloor$). *If $\Gamma;\vdash_{\mathtt{t}} a : \sigma$, then $\lfloor a \rfloor \rightarrow_\beta b$ iff $a \overset{*}{\rightarrow}_{\mathtt{cv}}\rightarrow_\beta c$ with $\lfloor c \rfloor = b$.*

$$\begin{tikzcd} a \arrow[r, "\mathtt{cv}*"] \arrow[d, mapsto] & {} \arrow[r, "\beta"] \arrow[d, phantom, "\Updownarrow"] & c \arrow[d, mapsto] \\ \lfloor a \rfloor \arrow[rr, "\beta"'] & & b \end{tikzcd}$$

Proof. The if part is given by Proposition 19. For the only if part we can suppose that $a = a_1 a_2$ with $\lfloor a_1 \rfloor = \lambda x.d$, so that $(\lambda x.d)\lfloor a_2 \rfloor$ is the redex fired in $\lfloor a \rfloor$, i.e. $b = d\,\{\lfloor a_2 \rfloor/x\}$. We can reduce to such a case reasoning by structural induction on a, discarding all the parts of the context where the reduction does not occur.

As a_1 is applied to a_2 there is a derivation giving $\Gamma';\vdash_{\mathtt{t}} a_1 : \tau \rightarrow \tau'$ for some Γ', τ, τ'. We can then $\mathtt{cv}$-normalize a_1 to a_1' (Corollary 13), which by subject reduction has the same type. Moreover by Proposition 19(iii) $\lfloor a_1' \rfloor = \lfloor a_1 \rfloor = \lambda x.d$, and we conclude by Lemma 20 that $a_1' = \lambda x.e$ with $\lfloor e \rfloor = d$, and we finally get $a_1 a_2 \overset{*}{\rightarrow}_{\mathtt{cv}} (\lambda x.e)a_2 \rightarrow_\beta e\,\{a_2/x\}$. Now by Lemma 17 $\lfloor e\,\{a_2/x\} \rfloor = \lfloor e \rfloor\,\{\lfloor a_2 \rfloor/x\} = d\,\{\lfloor a_2 \rfloor/x\} = b$ and we are done. □

Notice that the above result entails bisimulation with $\rightarrow_{\mathtt{c}}$ as a more general silent action: Proposition 19 gives the if part, while $\rightarrow_{\mathtt{cv}} \subseteq \rightarrow_{\mathtt{c}}$ gives the only if one.

4 The Translation

A translation from $\mathsf{xML^F}$ terms and instantiations into the coercion calculus is given in Figure 3. The idea is that instantiations can be seen as coercions; thus a term starting with a type abstraction $\Lambda(\alpha \geq \tau)$ becomes a term waiting for a coercion of type $\tau^\bullet \multimap \alpha$, and a term $a\phi$ is becomes a° coerced by ϕ°. The

[8] All the results shown so far are valid also without such a condition.

Types and contexts

$$\alpha^\bullet := \alpha, \qquad (\sigma \to \tau)^\bullet := \sigma^\bullet \to \tau^\bullet, \qquad (x : \tau)^\bullet := x : \tau^\bullet,$$
$$\bot^\bullet := \forall\alpha.\alpha, \quad (\forall(\alpha \geq \sigma)\tau)^\bullet := \forall\alpha.(\sigma^\bullet \multimap \alpha) \to \tau^\bullet, \quad (\alpha \geq \tau)^\bullet := i_\alpha : \tau^\bullet \multimap \alpha.$$

Instantiations

$$\tau^\circ := \underline{\lambda} x.x, \qquad (⅋)^\circ := \underline{\lambda} x.\underline{\Lambda} i_\alpha.x, \qquad (\phi;\psi)^\circ := \underline{\lambda} z.\psi^\circ \triangleright (\phi^\circ \triangleright z),$$
$$(!\alpha)^\circ := i_\alpha, \qquad (\&)^\circ := \underline{\lambda} x.x \triangleleft \underline{\lambda} z.z, \qquad (\mathbf{1})^\circ := \underline{\lambda} z.z,$$
$$(\forall(\geq \phi))^\circ := \underline{\lambda} x.\underline{\Lambda} i_\alpha.x \triangleleft (\underline{\lambda} z.i_\alpha \triangleright (\phi^\circ \triangleright z)),$$
$$(\forall(\alpha \geq)\phi)^\circ := \underline{\lambda} x.\underline{\Lambda} i_\alpha.\phi^\circ \triangleright (x \triangleleft i_\alpha).$$

Terms

$$x^\circ := x, \qquad (\lambda(x:\tau)a)^\circ := \lambda x.a^\circ, \qquad (ab)^\circ := a^\circ b^\circ,$$
$$(\Lambda(\alpha \geq \tau)a)^\circ := \underline{\Lambda} i_\alpha.a^\circ, \qquad (a\phi)^\circ := \phi^\circ \triangleright a^\circ.$$

Fig. 3: Translation of types, instantiations and terms into the coercion calculus. For every type variable α we suppose fixed a fresh term variable i_α.

rest of this section is devoted to showing how this translation and the properties of the coercion calculus lead to the main result of this work, SN of both $\mathsf{xML^F}$ and $\mathsf{eML^F}$. First one needs to show that the translation maps to typed terms. As expected, type instantiations are mapped to coercions.

proof in tech. app. ←

Lemma 22. *Let a be an $\mathsf{xML^F}$ term and ϕ be an instantiation:*

(i) if $\Gamma \vdash \phi : \sigma \leq \tau$ then $\Gamma^\bullet; \vdash_{\mathtt{c}} \phi^\circ : \sigma^\bullet \multimap \tau^\bullet$.
(ii) if $\Gamma \vdash a : \sigma$ then $\Gamma^\bullet; \vdash_{\mathtt{t}} a^\circ : \sigma^\bullet$.

With the substitution lemma below we will have simulation within reach.

proof in tech. app. ←

Lemma 23. *Let A be an $\mathsf{xML^F}$ term or an instantiation. Then we have:*

(i) $(A\{b/x\})^\circ = A^\circ\{b^\circ/x\}$,
(ii) $(A\{\mathbf{1}/!\alpha\}\{\tau/\alpha\})^\circ = A^\circ\{\underline{\lambda} z.z/i_\alpha\}$,
(iii) $(A\{\phi;!\alpha/!\alpha\})^\circ = A^\circ\{(\underline{\lambda} z.i_\alpha \triangleright (\phi^\circ \triangleright z))/i_\alpha\}$.

proof in tech. app. ←

Theorem 24 (Coercion calculus simulates $\mathsf{xML^F}$). *If $a \to_\beta b$ (resp. $a \to_\iota b$) in $\mathsf{xML^F}$, then $a^\circ \to_\beta b^\circ$ (resp. $a^\circ \xrightarrow{+}_{\mathtt{c}} b^\circ$) in coercion calculus.*

Proof. (Sketch) As the translation is contextual, it sufficies to analyze each reduction rule, perform the reductions and apply Lemma 23 where needed. □

Corollary 25 (Termination). *$\mathsf{xML^F}$ is strongly normalizing.*

The above already shows SN of $\mathsf{xML^F}$, however in order to prove that $\mathsf{eML^F}$ is also normalizing we need to make sure that ι-redexes cannot block β ones: in other words, a bisimulation result. We first need some technical lemmas, proved by structural induction. We recall that $\lceil a \rceil$ is the type erasure of a (page 3).

Lemma 26. *The type erasure of an $\mathsf{xML^F}$ term a coincides with the coercion erasure of its translation, i.e. $\lceil a \rceil = \lfloor a^\circ \rfloor$.*

Lemma 27.

proof in tech. app. ←

(i) If $a^\circ \to_\beta b$ then $a \to_\beta c$ with $c^\circ = b$;
(ii) if $a^\circ \to_{\mathtt{cv}} b$ then $a \to_\iota c$ with $b \overset{=}{\to}_{\mathtt{cv}} c^\circ$.

$$\begin{tikzcd} a \arrow[r, dashed, "\beta"] \arrow[d, mapsto] & c \arrow[d, mapsto] \\ a^\circ \arrow[r, "\beta"] & b \end{tikzcd} \qquad \begin{tikzcd} a \arrow[rr, dashed, "\iota"] \arrow[d, mapsto] & & c \arrow[d, mapsto] \\ a^\circ \arrow[r, "\mathtt{cv}"] & b \arrow[r, dashed, "\mathtt{cv}="] & c^\circ \end{tikzcd}$$

Notice that the above is not true in general for $\to_{\mathtt{c}}$ in place of $\to_{\mathtt{cv}}$: for example $x\&$ is normal in $\mathsf{xML^F}$, but $(x\&)^\circ = (\underline{\lambda} y.y) \triangleright x \to_{\mathtt{c}} x$.

The following lemma allows us lift to $\mathsf{xML^F}$ the reduction in coercion calculus that bisimulates β-steps (see Theorem 21).

Lemma 28 (Lifting). *Given a typed $\mathsf{xML^F}$ term a, we have that if $a^\circ \overset{*}{\to}_{\mathtt{cv}}\to_\beta b$ then $a \overset{*}{\to}_\iota\to_\beta c$ with $b \overset{*}{\to}_{\mathtt{c}} c^\circ$.*

$$\begin{tikzcd} a \arrow[rr, dashed, "\iota*"] \arrow[d, mapsto] & & \cdot \arrow[r, dashed, "\beta"] & c \arrow[d, mapsto] \\ a^\circ \arrow[r, "\mathtt{cv}*"] & \cdot \arrow[r, "\beta"] & b \arrow[r, dashed, "\mathtt{c}*"] & c^\circ \end{tikzcd}$$

Proof. As $\to_{\mathtt{cv}}$ is strongly normalizing (Corollary 13), we can reason by well-founded induction on a° with respect to $\to_{\mathtt{cv}}$.

First let us suppose that $a^\circ \to_\beta b$: we then apply Lemma 27(i) and get the result directly. Suppose then that $a^\circ \overset{+}{\to}_{\mathtt{cv}}\to_\beta b$. We have the following diagram:

$$\begin{tikzcd} a \arrow[r, dashed, "\iota"] \arrow[dd, mapsto] \arrow[ddr, phantom, "\text{(i)}"] & a_1 \arrow[rr, dashed, "\iota*"] \arrow[d, mapsto] \arrow[drrr, phantom, "\text{(iv)}"] & & \cdot \arrow[r, dashed, "\beta"] & c \arrow[d, mapsto, dashed] \\ & a_1^\circ \arrow[r, dashed, "\mathtt{cv}*"] & \cdot \arrow[r, dashed, "\beta"] & \cdot \arrow[r, dashed, "\mathtt{c}*"] & c^\circ \\ a^\circ \arrow[r, "\mathtt{cv}"'] & \cdot \arrow[u, dashed, "\mathtt{cv}="] \arrow[r, "\mathtt{cv}*"'] \arrow[ur, phantom, "\text{(ii)}"] & \cdot \arrow[u, dashed, "\mathtt{cv}*"] \arrow[r, "\beta"'] \arrow[ur, phantom, "\text{(iii)}"] & b \arrow[u, dashed, "\mathtt{c}*"'] & \end{tikzcd}$$

where (i) comes from Lemma 27(ii), (ii) is by confluence (Proposition 9), (iii) is by Proposition 19(i) and (iv) is by inductive hypothesis, as $a^\circ \overset{+}{\to}_{\mathtt{cv}} a_1^\circ$. ☐

Theorem 29 (Bisimulation of $\lceil . \rceil$). *Given a typed $\mathsf{xML^F}$ term a, we have that $\lceil a \rceil \to_\beta b$ iff $a \overset{*}{\to}_\iota\to_\beta c$ with $\lceil c \rceil = b$.*

$$\begin{tikzcd} a \arrow[r, "\iota*"] \arrow[d, mapsto] & \cdot \arrow[r, "\beta"] \arrow[d, phantom, "\Updownarrow"] & c \arrow[d, mapsto] \\ \lceil a \rceil \arrow[rr, "\beta"'] & & b \end{tikzcd}$$

Proof. For the if part, by Theorem 24 we have $a^\circ \overset{*}{\to}_{\mathtt{c}}\to_\beta c^\circ$, which by Lemma 26 and Proposition 19 implies $\lceil a \rceil = \lfloor a^\circ \rfloor \to_\beta \lfloor c^\circ \rfloor = \lceil c \rceil$. For the only if part, as $\lfloor a^\circ \rfloor = \lceil a \rceil \to_\beta b$, by Theorem 21 $a^\circ \overset{*}{\to}_{\mathtt{cv}}\to_\beta b'$ with $\lfloor b' \rfloor = b$. Now by Lemma 28 we have that $b' \overset{*}{\to}_{\mathtt{c}} c^\circ$ with $a \overset{*}{\to}_\iota\to_\beta c$. To conclude, we see that $\lceil c \rceil = \lfloor c^\circ \rfloor = \lfloor b' \rfloor = b$, where we used Lemma 26 and Proposition 19(iii). ☐

The above proof may be completely carried out within $\mathsf{xML^F}$, by applying a suitably modified version of Lemma 20. However, we preferred this formulation since it provides a better understanding of what happens on the side of the coercion calculus.

→ proof in tech. app.

Corollary 30. *Terms typed in $\mathsf{iML^F}$ and $\mathsf{eML^F}$ are strongly normalizing.*

Proof. Immediate by the above result and Theorem 2. ☐

Further work. We were able to prove new results for $\mathsf{ML^F}$ (namely SN and bisimulation of $\mathsf{xML^F}$ with its type erasure) by employing a more general calculus of coercions. It becomes natural then to ask whether its type system may be a framework to study coercions in general. A first natural target are the coercions arising from Leijen's translation of $\mathsf{ML^F}$ [5], which is more optimized than ours, in the sense that it does not add additional and unneeded structure to system F types. We plan then to study the coercions arising in F_η [8] or when using subtyping [9]. As explained at the start of section 3, $\mathsf{F_c}$ was purposely tailored down to suit $\mathsf{xML^F}$, stripping it of natural features.

A first, easy extension would consist in more liberal types and typing rules, allowing coercion polymorphism, coercion abstraction of coercions or even coercions between coercions (i.e. allowing types $\forall\alpha.\kappa$, $\kappa_1 \to \kappa_2$ and $\kappa_1 \multimap \kappa_2$). To progress further however, one would need a way to build coercions of arrow types, which are unneeded in $\mathsf{xML^F}$. Namely, given coercions $c_1 : \sigma_2 \multimap \sigma_1$ and $c_2 : \tau_1 \multimap \tau_2$, there should be a coercion $c_1 \Rightarrow c_2 : (\sigma_1 \to \tau_1) \multimap (\sigma_2 \to \tau_2)$, allowing a reduction $(c_1 \Rightarrow c_2) \triangleright \lambda x.a \to_{\mathsf{c}} \lambda x.c_2 \triangleright a\,\{c_1 \triangleright x/x\}$. This could be achieved either by introducing it as a primitive, by translation or by special typing rules. Indeed, if some sort of η-expansion would be available while building a coercion, one could write $c_1 \Rightarrow c_2 := \underline{\lambda} f.\lambda x.(c_2 \triangleright (f(c_1 \triangleright x)))$. However how to do this without loosing bisimulation is under investigation.

References

1. Milner, R., Tofte, M., Macqueen, D.: The Definition of Standard ML. MIT Press, Cambridge, MA, USA (1997)
2. Girard, J.Y., Lafont, Y., Taylor, P.: Proofs and Types. Number 7 in Cambridge tracts in theoretical computer science. Cambridge University Press (1989)
3. Le Botlan, D., Rémy, D.: MLF: Raising ML to the power of System F. In: Proc. of International Conference on Functional Programming (ICFP'03). (2003) 27–38
4. Rémy, D., Yakobowski, B.: A Church-style intermediate language for MLF. Submitted (July 2009)
5. Leijen, D.: A type directed translation of MLF to System F. In: Proc. of International Conference on Functional Programming (ICFP'07), ACM Press (2007)
6. Barber, A., Plotkin, G.: Dual intuitionistic linear logic. Technical report LFCS-96-347, University of Edinburgh (1997)
7. Baillot, P., Terui, K.: Light types for polynomial time computation in lambda calculus. Inf. Comput. **207**(1) (2009) 41–62
8. Mitchell, J.C.: Coercion and type inference. In: Proc. of 11^{th} symposium on Principles of programming languages (POPL'84), ACM (1984) 175–185
9. Crary, K.: Typed compilation of inclusive subtyping. In: Proc. of International Conference on Functional Programming (ICFP'00). (2000) 68–81

A Technical Proofs

This technical appendix is devoted to give full proofs of the results in the paper.

Lemma 5. *If $\Gamma; L \vdash_{\mathtt{c}\ell} a : \zeta$ then no subterm of a is of the form $\lambda x.b$ or bc. In particular a is β-normal.*

Proof. Let us call *regular* the terms of form $\lambda x.b$ or bc. We proceed by induction on the derivation of a. If $\Gamma; \vdash_{\mathtt{c}} a : \sigma \multimap \tau$ then the last rule is either Ax (in which case a is a variable and the result follows) or LAbs from $\Gamma; z : \sigma \vdash_\ell a' : \tau$ with $a = \underline{\lambda} z.a'$. Inductive hypothesis yields that no strict subterm of a (i.e. no subterm of a') is regular.

If $\Gamma; z : \sigma \vdash_\ell a : \tau$ then we reason by cases on the last rule. If it is LAx then $a = w$ and we are done; in all other cases it is sufficient to note that:

- a is not regular, and
- the premise or both the premises of the rule are of one of the two forms, so inductive hypothesis applies to every immediate subterm(s). □

Lemma 7 (Weakening and substitution). *We have the following:*

(i) $\Gamma; L \vdash_{\mathtt{tc}\ell} a : \zeta$ *and* $x \notin \mathrm{dom}(\Gamma; L)$ *entail* $\Gamma, x : \zeta'; L \vdash_{\mathtt{tc}\ell} a : \zeta$*;*
(ii) $\Gamma; \vdash_{\mathtt{tc}} a : \zeta'$ *and* $\Gamma, x : \zeta'; L \vdash_{\mathtt{tc}\ell} b : \zeta$ *entail* $\Gamma; L \vdash_{\mathtt{tc}\ell} b\,\{a/x\} : \zeta$*;*
(iii) $\Gamma; L \vdash_{\mathtt{t}\ell} a : \sigma$ *and* $\Gamma; x : \sigma \vdash_\ell b : \zeta$ *entail* $\Gamma; L \vdash_{\mathtt{t}\ell} b\,\{a/x\} : \zeta$*.*

Proof. The weakening result is obtained by a trivial induction on the size of the derivation. As usual, one may have to change the bound variable in the Gen rule.

For the substitution results, both are obtained by induction on the size of the derivation for b, by cases on its last rule.

- Ax: for (ii), if $b = x$ then the derivation of a is what looked for, as $\zeta' = \zeta$ and $b\,\{a/x\} = a$; otherwise $b\,\{a/x\} = b$ and we are done; (ii) does not happen.
- LAx: for (ii) $L = z : \sigma$ and $b = z \neq x$, so $\Gamma; z : \sigma \vdash_\ell z = z\,\{a/x\} : \sigma$ and we are done; for (iii) necessarily $b = x$, $\zeta = \sigma$ and $b\,\{a/x\} = a$ and we are done.
- Abs, App and LAbs: trivial application of inductive hypothesis for (ii), while it does not apply for (iii) as the judgment for b cannot be a linear one.
- CAbs, Gen and Inst: for these unary rules both (ii) and (iii) are trivial.
- CApp and LApp: for (ii) the substitution distributes as usual; for (iii) it must be noted that x does not appear free in one of the two subterms (as it does not appear in the assignment). Indeed we will have $(b_1 \triangleleft b_2)\,\{a/x\} = (b_1\,\{a/x\}) \triangleleft b_2$ (resp. $(b_1 \triangleright b_2)\,\{a/x\} = b_1 \triangleright (b_2\,\{a/x\})$) and inductive hypothesis is needed for just one of the two branches. □

The following standard lemma is used in some of the following results.

Lemma 31. *If $\Gamma; L \vdash_{\mathtt{tc}\ell} a : \zeta$, then there is a derivation of the same judgment where no* Inst *rule follows immediately a* Gen *one.*

Proof. One uses the following remark: if we have a derivation π of $\Gamma; L \vdash_{\mathtt{tc}\ell} a : \zeta$ then for any τ there is a derivation of the same size, which we will denote by $\pi\,\{\tau/\alpha\}$, giving $\Gamma\,\{\tau/\alpha\}; L\,\{\tau/\alpha\} \vdash_{\mathtt{tc}\ell} a : \zeta\,\{\tau/\alpha\}$. To show it, it suffices to substitute τ for all α's, possibly renaming bound variables along the process.

One then shows this standard result by structural induction on the size of the derivation π of $\Gamma; L \vdash_{\mathtt{tc}\ell} a : \zeta$. Suppose in fact that there is an Inst rule immediately after a Gen one. Then there is a subderivation π' of the following shape:

$$\dfrac{\dfrac{\begin{array}{c}\pi'' \\ \vdots \\ \Gamma'; L' \vdash_{\mathtt{t}\ell} b : \sigma\end{array} \qquad \alpha \notin \mathrm{ftv}(\Gamma'; L')}{\Gamma'; L' \vdash_{\mathtt{t}\ell} b : \forall\alpha.\sigma}\,\textsc{Gen}}{\Gamma'; L' \vdash_{\mathtt{t}\ell} b : \sigma\,\{\tau/\alpha\}}\,\textsc{Inst}$$

By applying the above remark it suffices to substitute π' in π with $\pi''\,\{\tau/\alpha\}$, as $\Gamma'\,\{\tau/\alpha\}\,; L'\,\{\tau/\alpha\} = \Gamma'; L'$. The derivation thus obtained is smaller by two rules, so inductive hypothesis applies and we are done. ☐

Proposition 8 (Subject reduction). *If $a \to_{\beta\mathtt{c}} b$ and $\Gamma; L \vdash_{\mathtt{t}\ell\mathtt{c}} a : \zeta$, then $\Gamma; L \vdash_{\mathtt{t}\ell\mathtt{c}} b : \zeta$.*

Proof. By Lemma 31 we can suppose that in the derivation of $a : \zeta$ there is no Inst rule immediately following a Gen. One then reasons by induction on the size of the derivation to settle the context closure, stripping the cases down to when the last rule of the derivation is one of the application rules App, CApp or LApp which introduces the redex $(\lambda x.c)d$, $(\underline{\measuredangle} x.c) \lhd d$ or $(\underline{\lambda} x.c) \rhd d$. Moreover we can see that no Gen or Inst rule is present between the abstraction rule and the application one: if there were any, then as no Inst follows Gen we would have a sequence of Inst rules followed by Gen ones. However the former cannot follow an abstraction, while the latter cannot precede an application on the function side.

- $(\lambda x.c)d \to_\beta c\,\{d/x\}$: then $\Gamma, x : \sigma; \vdash_{\mathtt{t}} c : \tau$ and $\Gamma; \vdash_{\mathtt{t}} d : \sigma$, and Lemma 7(ii) settles the case;
- $(\underline{\measuredangle} x.c) \lhd d \to_{\mathtt{c}} c\,\{d/x\}$: the rule introducing $\underline{\measuredangle} x.c$ must be CAbs, with $\Gamma, x : \kappa; L \vdash_{\mathtt{t}\ell} c : \sigma$ and $\Gamma; \vdash_{\mathtt{c}} d : \kappa$, and again Lemma 7(ii) entails the result;
- $(\underline{\lambda} x.c) \rhd d \to_{\mathtt{c}} c\,\{d/x\}$: here $\underline{\lambda} x.c$ is introduced by LAbs, so $\Gamma; x : \tau \vdash_\ell c : \sigma$ and $\Gamma; L \vdash_{\mathtt{t}\ell} d : \tau$, and it is Lemma 7(iii) that applies. ☐

Lemma 15.

(i) If $\Gamma, x : \kappa; L \vdash_{\mathtt{t}\ell} a : \sigma$ then $x \notin \mathrm{fv}(\lfloor a \rfloor)$;
(ii) if $\Gamma; z : \tau \vdash_\ell a : \sigma$ then $\lfloor a \rfloor = z$.

Proof. Both are proved by induction on the derivation, by cases on the last rule.

(i) As the judgment is not a coercion one, Ax cannot yield $a = x$, nor can LAx. Inductive hypothesis applies seamlessly for rules Abs, App, CAbs, Gen and Inst. The LAbs rule cannot be the last one of the derivation. Finally, rule CApp (resp. LApp) gives $\lfloor a \rfloor = \lfloor b \lhd c \rfloor = \lfloor b \rfloor$ (resp. $\lfloor a \rfloor = \lfloor b \rhd c \rfloor = \lfloor c \rfloor$), and inductive hypothesis applied to the left (resp. right) branch gives the result.

(ii) The judgment is required to be a linear one: Ax, Abs, App and LAbs do not apply. For LAx we have $a = w$ and we are done. For all the other rules the result follows by inductive hypothesis, possibly chasing the $\Gamma; z : \tau$ environment left or right in the CApp and LApp rules respectively. ☐

Lemma 18. *If $\Gamma; x : \tau \vdash_\ell a : \sigma$ and $b \to_\beta c$, then $a\,\{b/x\} \to_\beta a\,\{c/x\}$.*

Proof. By induction on the derivation, by cases on the last rule used: Ax, Abs, App and LAbs do not apply; LAx is trivial (as $a = x$); in CAbs, Gen and Inst the inductive hypothesis easily yields the inductive step; finally in CApp and LApp the inductive hypothesis is applied only to the left and right premises respectively, giving the needed one step by context closure. ☐

Proposition 19. *Suppose that a is an F_{c} term. Then:*

(i) if $b_1 \leftarrow_{\mathsf{c}} a \to_\beta b_2$ then there is c with $b_1 \to_\beta c \overset{*}{\leftarrow}_{\mathsf{c}} b_2$;
(ii) if $a \to_\beta b$ then $\lfloor a \rfloor \to \lfloor b \rfloor$;
(iii) if $a \to_{\mathsf{c}} b$ then $\lfloor a \rfloor = \lfloor b \rfloor$.

$$\begin{tikzcd} a \arrow[r, "\beta"] \arrow[d, "\mathsf{c}"'] & b_2 \arrow[d, dashed, "\mathsf{c}*"] \\ b_1 \arrow[r, dashed, "\beta"] & c \end{tikzcd}$$

Proof.

(i) We consider the case where the two redexes are not orthogonal: by non-overlapping one contains the other, and we can suppose that a is the biggest of the two, closing the diagram by context in the other cases.
If $a = (\lambda x.d)e$, then the diagram is closed straightforwardly, whether the c-redex is in d or in e (in which case many or no c-steps may be needed).
When firing $a = (\underline{\lambda} x.d) \triangleright e$ then by typing $\underline{\lambda} x.d$ is a coercion, so we have a derivation ending in $\Gamma; x : \sigma \vdash_\ell d : \tau$, with $\Gamma; \vdash_{\mathsf{t}} e : \sigma$ (we are silently using Lemma 31 here). As d cannot contain any β-redex, the other redex fired in the diagram is in e, so $e \to_\beta e'$. Thus $b_1 = d\,\{e/x\}$ and $b_2 = (\underline{\lambda} x.d) \triangleright e' \to_{\mathsf{c}} d\,\{e'/x\}$. By Lemma 18 we have that $b_1 \to_\beta d\,\{e'/x\}$ and we are done.
If firing $a = (\underline{\reflectbox{\ensuremath{\lambda}}} x.d) \triangleleft e$ we have that e is a coercion, which cannot contain any β-redex, so we have $d \to_\beta d'$ and $b_2 = (\underline{\reflectbox{\ensuremath{\lambda}}} x.d') \triangleleft e$. We easily get $b_2 \to_{\mathsf{c}} d'\,\{e/x\} \leftarrow_\beta d\,\{e/x\} = b_1$.
(ii) By Lemma 17, as $\lfloor (\lambda x.c)d \rfloor = (\lambda x.\lfloor c \rfloor)\lfloor d \rfloor \to \lfloor c \rfloor\,\{\lfloor d \rfloor/x\} = \lfloor c\,\{d/x\} \rfloor$.
(iii) Proceeding by context closure, suppose $a = (\underline{\reflectbox{\ensuremath{\lambda}}} x.c) \triangleleft d$ (resp. $a = (\underline{\lambda} x.c) \triangleright d$), so $b = c\,\{d/x\}$. In the first case we will have $\lfloor a \rfloor = \lfloor c \rfloor$ and $\Gamma, x : \kappa; L \vdash_{\mathsf{t}\ell} c : \sigma$ for some typing derivation. Then by Lemmas 15(i) and 17 we have that $x \notin \mathrm{fv}(\lfloor c \rfloor)$ and $\lfloor b \rfloor = \lfloor c \rfloor\,\{\lfloor d \rfloor/x\} = \lfloor c \rfloor = \lfloor a \rfloor$ and we are done.
In the latter case we have $\lfloor a \rfloor = \lfloor d \rfloor$, and $\Gamma; x : \tau \vdash_\ell c : \sigma$. Lemmas 15(ii) and 17 entail $\lfloor b \rfloor = \lfloor c \rfloor\,\{\lfloor d \rfloor/x\} = x\,\{\lfloor d \rfloor/x\} = \lfloor d \rfloor = \lfloor a \rfloor$ and we are again done. ☐

Lemma 20. *Every typable cv-normal term a such that $\lfloor a \rfloor = \lambda x.b$ is a c-value. In particular if a has an arrow type then $a = \lambda x.c$ with $\lfloor c \rfloor = b$.*

Proof. We reason by structural induction on a. Notice a cannot be a variable or a regular application, as its erasure is an abstraction. Following are the remaining cases.

- $a = \lambda y.d$: a is a c-value.
- $a = \underline{\measuredangle} y.d$: as $\lfloor d \rfloor = \lfloor a \rfloor = \lambda x.b$ inductive hypothesis applies and d is a c-value, hence a is a c-value too.
- $a = \underline{\lambda} y.d$: this case cannot happen, as no coercion has an abstraction as erasure.
- $a = d \triangleleft e$: by inductive hypothesis ($\lfloor d \rfloor = \lfloor a \rfloor = \lambda x.b$) we have that d is a c-value. We arrive to a contradiction ruling out all the alternatives for d:
  - $d = \lambda y.f$ would make $d \triangleright e$ impossible to type;
  - $d = \underline{\measuredangle} y.f$ with f a c-value is impossible as $d \triangleright e$ would be a valid cv-redex;
  - $d = x \triangleright f$ with f a c-value is impossible as, before the CApp introducing $d \triangleright e$, d would be typed by a type variable α (as x would necessarily have type $\sigma \multimap \alpha$), which in no way could lead to the necessary type $\kappa \to \tau$.
- $a = d \triangleright e$: by inductive hypothesis ($\lfloor e \rfloor = \lfloor a \rfloor = \lambda x.b$) e is a c-value. As d is a coercion, by Remark 4 it can either be a variable (in which case we are done) or an abstraction. The latter however is impossible as a would be a valid cv-redex.

For the consequence about an arrow-typed a, it suffices to see that $\underline{\measuredangle} y.u$ gives rise to a (possibly generalized) type $\kappa \to \tau$, while $x \triangleright u$ gives a (not generalizable) type variable. So in this case the only possibility for a is to be an abstraction $\lambda x.c$. The fact that $\lfloor c \rfloor = b$ follows from the definition of $\lfloor a \rfloor$. ☐

The following lemma is needed for proving Lemma 22 below.

Lemma 32. $(\sigma\,\{\tau/\alpha\})^\bullet = \sigma^\bullet\,\{\tau^\bullet/\alpha\}$.

Proof. By structural induction on σ.

- $\sigma = \alpha$: $(\alpha\,\{\tau/\alpha\})^\bullet = \tau^\bullet = \alpha^\bullet\,\{\tau^\bullet/\alpha\}$.
- $\sigma = \beta \neq \alpha$: $(\beta\,\{\tau/\alpha\})^\bullet = \beta^\bullet = \beta^\bullet\,\{\tau^\bullet/\alpha\}$.
- $\sigma = \sigma_1 \to \sigma_2$: we have $((\sigma_1 \to \sigma_2)\,\{\tau/\alpha\})^\bullet = (\sigma_1\,\{\tau/\alpha\} \to \sigma_2\,\{\tau/\alpha\})^\bullet = (\sigma_1\,\{\tau/\alpha\})^\bullet \to (\sigma_2\,\{\tau/\alpha\})^\bullet$. By the induction hypothesis, this is equal to $\sigma_1^\bullet\,\{\tau^\bullet/\alpha\} \to \sigma_2^\bullet\,\{\tau^\bullet/\alpha\} = (\sigma_1^\bullet \to \sigma_2^\bullet)\,\{\tau^\bullet/\alpha\}$.
- $\sigma = \bot$: $(\bot\,\{\tau/\alpha\})^\bullet = \bot^\bullet = \forall\beta.\beta = (\forall\beta.\beta)\,\{\tau^\bullet/\alpha\} = \bot^\bullet\,\{\tau^\bullet/\alpha\}$.
- $\sigma = \forall\beta \geq \sigma_1.\sigma_2$ (supposing $\beta \notin \mathrm{ftv}(\tau) \cup \{\alpha\}$):

$$\begin{aligned}
((\forall\beta \geq \sigma_1.\sigma_2)\,\{\tau/\alpha\})^\bullet &= (\forall\beta \geq \sigma_1\,\{\tau/\alpha\}\,.\sigma_2\,\{\tau/\alpha\})^\bullet \\
&= \forall\beta.((\sigma_1\,\{\tau/\alpha\})^\bullet \multimap \beta) \to \sigma_2^\bullet\,\{\tau^\bullet/\alpha\} \\
&= \forall\beta.(\sigma_1^\bullet\,\{\tau^\bullet/\alpha\} \multimap \beta) \to \sigma_2^\bullet\,\{\tau^\bullet/\alpha\} \\
&= (\forall\beta.(\sigma_1^\bullet \multimap \beta) \to \sigma_2^\bullet)\,\{\tau^\bullet/\alpha\} = (\forall\beta \geq \sigma_1.\sigma_2)^\bullet\,\{\tau^\bullet/\alpha\}
\end{aligned}$$

  where we applied inductive hypothesis for the third equality. ☐

Lemma 22. *Let a be an $\mathsf{xML^F}$ term and ϕ be an instantiation:*

(i) if $\Gamma \vdash \phi : \sigma \leq \tau$ then $\Gamma^\bullet;\vdash_{\mathtt{c}} \phi^\circ : \sigma^\bullet \multimap \tau^\bullet$.
(ii) if $\Gamma \vdash a : \sigma$ then $\Gamma^\bullet;\vdash_{\mathtt{t}} a^\circ : \sigma^\bullet$.

Proof. (i) By induction on the derivation of $\Gamma \vdash \phi : \sigma \leq \tau$.

– IBot, $\Gamma \vdash \tau : \bot \leq \tau$. We have to prove that $\Gamma^\bullet;\vdash_{\mathsf{c}} \underline{\lambda} x.x : (\forall\alpha.\alpha) \multimap \tau^\bullet$. This follows by applying LAbs, Inst and LAx.

– IAbstr, $\Gamma \vdash\, !\alpha : \tau \leq \alpha$ where $\alpha \geq \tau \in \Gamma$. We have to prove $\Gamma^\bullet;\vdash_{\mathsf{c}} i_\alpha : \tau^\bullet \multimap \alpha$, which follows from Ax since $i_\alpha : \tau^\bullet \multimap \alpha \in \Gamma^\bullet$.

– IUnder, $\Gamma \vdash \forall(\alpha \geq)\phi : \forall(\alpha \geq \sigma)\tau_1 \leq \forall(\alpha \geq \sigma)\tau_2$. By induction hypothesis we have a proof π of $\Gamma';\vdash_{\mathsf{c}} \phi^\circ : \tau_1^\bullet \multimap \tau_2^\bullet$ where $\Gamma' := \Gamma^\bullet, i_\alpha : \sigma^\bullet \multimap \alpha$. Let $L := x : \forall\alpha.(\sigma^\bullet \multimap \alpha) \to \tau_1^\bullet$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\pi\\ \vdots\\ \Gamma';\vdash_{\mathsf{c}} \phi^\circ : \tau_1^\bullet \multimap \tau_2^\bullet\end{array} \qquad \begin{array}{c}\dfrac{\dfrac{\dfrac{}{\Gamma';L\vdash_\ell x : (\forall(\alpha\geq\sigma)\tau_1)^\bullet}\textsc{LAx}}{\Gamma';L\vdash_\ell x : (\sigma^\bullet \multimap \alpha)\to\tau_1^\bullet}\textsc{Inst} \qquad \dfrac{}{\Gamma';\vdash_{\mathsf{c}} i_\alpha : \sigma^\bullet\multimap\alpha}\textsc{Ax}}{}\textsc{CApp}\\ \vdots\\ \Gamma';L\vdash_\ell x \triangleleft i_\alpha : \tau_1^\bullet\end{array}}{\Gamma';L\vdash_\ell \phi^\circ \triangleright (x\triangleleft i_\alpha) : \tau_2^\bullet}\textsc{LApp}}{\Gamma^\bullet;L\vdash_\ell \underline{\Lambda} i_\alpha.\phi^\circ \triangleright (x\triangleleft i_\alpha) : (\sigma^\bullet\multimap\alpha)\to\tau_2^\bullet}\textsc{CAbs}}{\Gamma^\bullet;L\vdash_\ell \underline{\Lambda} i_\alpha.\phi^\circ \triangleright (x\triangleleft i_\alpha) : \forall\alpha.(\sigma^\bullet\multimap\alpha)\to\tau_2^\bullet}\textsc{Gen}}{\Gamma^\bullet;\vdash_{\mathsf{c}} \underline{\lambda} x.\underline{\Lambda} i_\alpha.\phi^\circ \triangleright (x\triangleleft i_\alpha) : (\forall(\alpha\geq\sigma)\tau_1)^\bullet \multimap (\forall(\alpha\geq\sigma)\tau_2)^\bullet}\textsc{LAbs}$$

– IComp, $\Gamma \vdash \phi;\psi : \tau_1 \leq \tau_3$. By induction hypothesis we have a proof π_1 of $\Gamma^\bullet;\vdash_{\mathsf{c}} \phi^\circ : \tau_1^\bullet \multimap \tau_2^\bullet$, and a proof π_2 of $\Gamma^\bullet;\vdash_{\mathsf{c}} \psi^\circ : \tau_2^\bullet \multimap \tau_3^\bullet$. Then we can build the following proof:

$$\dfrac{\dfrac{\begin{array}{c}\pi_2\\ \vdots\\ \Gamma^\bullet;\vdash_{\mathsf{c}} \psi^\circ : \tau_2^\bullet \multimap \tau_3^\bullet\end{array} \qquad \dfrac{\begin{array}{c}\pi_1\\ \vdots\\ \Gamma^\bullet;\vdash_{\mathsf{c}} \phi^\circ : \tau_1^\bullet \multimap \tau_2^\bullet\end{array} \qquad \dfrac{}{\Gamma^\bullet; z:\tau_1^\bullet \vdash_\ell z : \tau_1^\bullet}\textsc{LAx}}{\Gamma^\bullet; z:\tau_1^\bullet \vdash_\ell \phi^\circ \triangleright z : \tau_2^\bullet}\textsc{LApp}}{\Gamma^\bullet; z:\tau_1^\bullet \vdash_\ell \psi^\circ \triangleright (\phi^\circ \triangleright z) : \tau_3^\bullet}\textsc{LApp}}{\Gamma^\bullet;\vdash_{\mathsf{c}} \underline{\lambda} z.\psi^\circ \triangleright (\phi^\circ \triangleright z) : \tau_1^\bullet \multimap \tau_3^\bullet}\textsc{LAbs}$$

– IInside, $\Gamma \vdash \forall(\geq \phi) : \forall(\alpha \geq \tau_1)\sigma \leq \forall(\alpha \geq \tau_2)\sigma$. We can suppose $\alpha \notin \mathrm{ftv}(\Gamma) = \mathrm{ftv}(\Gamma^\bullet)$. We set $L := x : (\forall(\alpha \geq \tau_1)\sigma)^\bullet$ and $\Gamma' := \Gamma^\bullet, i_\alpha : (\tau_2^\bullet \multimap \alpha)$. By induction hypothesis (and Lemma 7(i)) we have a proof of $\Gamma';\vdash_{\mathsf{c}} \phi^\circ : \tau_1^\bullet \multimap \tau_2^\bullet$. By mixing it with $\Gamma';\vdash_{\mathsf{c}} i_\alpha : \tau_2^\bullet \multimap \alpha$ and going through the same derivation as above for IComp, we get a proof π of $\Gamma';\vdash_{\mathsf{c}} \underline{\lambda} z.i_\alpha \triangleright (\phi^\circ \triangleright z) : \tau_1^\bullet \multimap \alpha$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma';L\vdash_\ell x : (\forall(\alpha\geq\tau_1)\sigma)^\bullet}\textsc{LAx}}{\Gamma';L\vdash_\ell x : (\tau_1^\bullet\multimap\alpha)\to\sigma^\bullet}\textsc{Inst} \qquad \begin{array}{c}\pi\\ \vdots\\ \Gamma';\vdash_{\mathsf{c}} \underline{\lambda} z.i_\alpha \triangleright (\phi^\circ \triangleright z) : \tau_1^\bullet \multimap \alpha\end{array}}{\Gamma';L\vdash_\ell x \triangleleft (\underline{\lambda} z.i_\alpha \triangleright (\phi^\circ \triangleright z)) : \sigma^\bullet}\textsc{CApp}}{\Gamma^\bullet;L\vdash_\ell \underline{\Lambda} i_\alpha.x \triangleleft (\underline{\lambda} z.i_\alpha \triangleright (\phi^\circ \triangleright z)) : (\tau_2^\bullet\multimap\alpha)\to\sigma^\bullet}\textsc{CAbs}}{\Gamma^\bullet;L\vdash_\ell \underline{\Lambda} i_\alpha.x \triangleleft (\underline{\lambda} z.i_\alpha \triangleright (\phi^\circ \triangleright z)) : (\forall(\alpha\geq\tau_2)\sigma)^\bullet}\textsc{Gen}}{\Gamma^\bullet;\vdash_{\mathsf{c}} \underline{\lambda} x.\underline{\Lambda} i_\alpha.x \triangleleft (\underline{\lambda} z.i_\alpha \triangleright (\phi^\circ \triangleright z)) : (\forall(\alpha\geq\tau_1)\sigma)^\bullet \multimap (\forall(\alpha\geq\tau_2)\sigma)^\bullet}\textsc{LAbs}$$

– IIntro, $\Gamma \vdash ⅋ : \tau \leq \forall(\alpha \geq \bot)\tau$ where $\alpha \notin \mathrm{ftv}(\tau)$. By α-conversion we can choose any $\alpha \notin \mathrm{ftv}(\Gamma^\bullet; x : \tau^\bullet)$, so the Gen rule in the following proof is applicable:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma^\bullet, i_\alpha : (\forall\beta.\beta) \multimap \alpha; x : \tau^\bullet \vdash_\ell x : \tau^\bullet}\,\textsc{LAx}}{\Gamma^\bullet; x : \tau^\bullet \vdash_\ell \underline{\not\lambda} i_\alpha.x : ((\forall\beta.\beta) \multimap \alpha) \to \tau^\bullet}\,\textsc{CAbs}}{\Gamma^\bullet; x : \tau^\bullet \vdash_\ell \underline{\not\lambda} i_\alpha.x : (\forall(\alpha \geq \bot)\tau)^\bullet}\,\textsc{Gen}}{\Gamma^\bullet; \vdash_{\mathtt{c}} \underline{\lambda} x.\underline{\not\lambda} i_\alpha.x : \tau^\bullet \multimap (\forall(\alpha \geq \bot)\tau)^\bullet}\,\textsc{LAbs}$$

– IElim, $\Gamma \vdash \& : \forall(\alpha \geq \sigma)\tau \leq \sigma\,\{\tau/\alpha\}$. Note that α can be chosen not in $\mathrm{ftv}(\sigma^\bullet)$ and that $(\tau\,\{\sigma/\alpha\})^\bullet = \tau^\bullet\,\{\sigma^\bullet/\alpha\}$ holds by Lemma 32. Let $L := x : \forall\alpha.(\sigma^\bullet \multimap \alpha) \to \tau^\bullet$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma^\bullet; L \vdash_\ell x : \forall\alpha.(\sigma^\bullet \multimap \alpha) \to \tau^\bullet}\,\textsc{LAx}}{\Gamma^\bullet; L \vdash_\ell x : (\sigma^\bullet \multimap \sigma^\bullet) \to \tau^\bullet\,\{\sigma^\bullet/\alpha\}}\,\textsc{Inst} \qquad \dfrac{\dfrac{}{\Gamma^\bullet; z : \sigma^\bullet \vdash_\ell z : \sigma^\bullet}\,\textsc{LAx}}{\Gamma^\bullet; \vdash_{\mathtt{c}} \underline{\lambda} z.z : \sigma^\bullet \multimap \sigma^\bullet}\,\textsc{LAbs}}{\Gamma^\bullet; L \vdash_\ell x \lhd \underline{\lambda} z.z : \tau^\bullet\,\{\sigma^\bullet/\alpha\}}\,\textsc{CApp}}{\Gamma^\bullet; \vdash_{\mathtt{c}} \underline{\lambda} x.x \lhd \underline{\lambda} z.z : (\forall(\alpha \geq \sigma)\tau)^\bullet \multimap (\tau\,\{\sigma/\alpha\})^\bullet}\,\textsc{LAbs}$$

– IId, $\Gamma \vdash \mathbf{1} : \tau \leq \tau$. We have $\Gamma^\bullet; \vdash_{\mathtt{c}} \underline{\lambda} z.z : \tau^\bullet \multimap \tau^\bullet$ by LAbs and LAx.

(ii) By induction on the derivation of $\Gamma \vdash a : \sigma$.

– Var, $\Gamma \vdash x : \tau$, where $\Gamma(x) = \tau$. We then get $\Gamma^\bullet; \vdash_{\mathtt{t}} x : \tau^\bullet$ by Ax.
– Abs, $\Gamma \vdash \lambda(x : \tau)a : \tau \to \sigma$. By induction hypothesis we have a proof of $\Gamma^\bullet, x : \tau^\bullet; \vdash_{\mathtt{t}} a : \sigma^\bullet$ which by Abs gives $\Gamma^\bullet; \vdash_{\mathtt{t}} \lambda x.a : \tau^\bullet \to \sigma^\bullet$.
– App, $\Gamma \vdash ab : \tau$. By induction hypothesis we have proofs for $\Gamma^\bullet; \vdash_{\mathtt{t}} a : \tau^\bullet \to \sigma^\bullet$ and π_2 of $\Gamma^\bullet; \vdash_{\mathtt{t}} b : \tau^\bullet$ giving $\Gamma^\bullet; \vdash_{\mathtt{t}} ab : \sigma^\bullet$ by App.
– TAbs, $\Gamma \vdash \Lambda(\alpha \geq \sigma)a : \forall(\alpha \geq \sigma)\tau$ where $\alpha \notin \mathrm{ftv}(\Gamma)$. It follows that $\alpha \notin \mathrm{ftv}(\Gamma^\bullet)$, and as by induction hypothesis we have a proof π of $\Gamma^\bullet, i_\alpha : \sigma^\bullet \multimap \alpha; \vdash_{\mathtt{t}} a^\circ : \tau^\bullet$ we have

$$\dfrac{\dfrac{\begin{array}{c}\pi\\ \vdots\\ \Gamma^\bullet, i_\alpha : \sigma^\bullet \multimap \alpha; \vdash_{\mathtt{t}} a^\circ : \tau^\bullet\end{array}}{\Gamma^\bullet; \vdash_{\mathtt{t}} \underline{\not\lambda} i_\alpha.a^\circ : (\sigma^\bullet \multimap \alpha) \to \tau^\bullet}\,\textsc{CAbs}}{\Gamma^\bullet; \vdash_{\mathtt{t}} \underline{\not\lambda} i_\alpha.a^\circ : \forall\alpha.(\sigma^\bullet \multimap \alpha) \to \tau^\bullet}\,\textsc{Gen}$$

– TApp, $\Gamma \vdash a\phi : \sigma$. Since $\Gamma \vdash \phi : \tau \leq \sigma$ holds we have a proof of $\Gamma^\bullet; \vdash_{\mathtt{c}} \phi^\circ : \tau^\bullet \multimap \sigma^\bullet$ by point (i) of this lemma. By induction hypothesis we have also a proof of $\Gamma^\bullet; \vdash_{\mathtt{t}} a^\circ : \tau^\bullet$. The two together combined with a LApp rule give $\Gamma^\bullet; \vdash_{\mathtt{t}} \phi^\circ \rhd a^\circ : \sigma^\bullet$. $\square$

Lemma 23. *Let A be a term or an instantiation. Then we have:*

(i) $(A\,\{b/x\})^\circ = A^\circ\,\{b^\circ/x\}$,
(ii) $(A\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\})^\circ = A^\circ\,\{\underline{\lambda} z.z/i_\alpha\}$,
(iii) $(A\,\{\phi; !\alpha/!\alpha\})^\circ = A^\circ\,\{(\underline{\lambda} z.i_\alpha \rhd (\phi^\circ \rhd z))/i_\alpha\}$.

Proof. All three results are carried out by structural induction on A. The inductive steps of (i) are straightforward, taking into account that if $A = \phi$ then $\phi\,\{b/x\} = \phi$.

For (ii), when A is a term the inductive step is immediate. Otherwise:

- $A = \sigma$: we have $(\sigma\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\})^\circ = (\sigma\,\{\tau/\alpha\})^\circ = \underline{\lambda}x.x$, which is equal to $(\underline{\lambda}x.x)\,\{\underline{\lambda}z.z/i_\alpha\} = \sigma^\circ\,\{\underline{\lambda}z.z/i_\alpha\}$.
- $A = !\alpha$: we have $(!\alpha\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\})^\circ = (\mathbf{1})^\circ = \underline{\lambda}z.z = i_\alpha\,\{\underline{\lambda}z.z/i_\alpha\} = (!\alpha)^\circ\,\{\underline{\lambda}z.z/i_\alpha\}$.
- $A = \forall(\geq \phi)$: we have

$$\begin{aligned}
(\forall(\geq \phi)\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\})^\circ &= (\forall(\geq \phi\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\}))^\circ \\
&= \underline{\lambda}x.\underline{\Lambda}i_\beta.x \triangleleft (\underline{\lambda}z.i_\beta \triangleright ((\phi\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\})^\circ \triangleright z)) \\
\text{(inductive hypothesis)} &= \underline{\lambda}x.\underline{\Lambda}i_\beta.x \triangleleft (\underline{\lambda}z.i_\beta \triangleright ((\phi^\circ\,\{\underline{\lambda}z.z/i_\alpha\}) \triangleright z)) \\
&= (\underline{\lambda}x.\underline{\Lambda}i_\beta.x \triangleleft (\underline{\lambda}z.i_\beta \triangleright (\phi^\circ \triangleright z)))\,\{\underline{\lambda}z.z/i_\alpha\} \\
&= (\forall(\geq \phi))^\circ\,\{\underline{\lambda}z.z/i_\alpha\}\,.
\end{aligned}$$

- $A = \forall(\beta \geq)\phi$: we have (supposing $\beta \notin \mathrm{ftv}(\tau) \cup \{\alpha\}$):

$$\begin{aligned}
((\forall(\beta \geq)\phi)\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\})^\circ &= (\forall(\beta \geq)\phi\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\})^\circ \\
&= \underline{\lambda}z.i_\beta.(\phi\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\})^\circ \triangleright (x \triangleleft i_\beta) \\
\text{(inductive hypothesis)} &= \underline{\lambda}z.i_\beta.(\phi^\circ\,\{\underline{\lambda}z.z/i_\alpha\}) \triangleright (x \triangleleft i_\beta) \\
&= (\underline{\lambda}z.i_\beta.\phi^\circ \triangleright (x \triangleleft i_\beta))\,\{\underline{\lambda}z.z/i_\alpha\} \\
&= (\forall(\beta \geq)\phi)^\circ\,\{\underline{\lambda}z.z/i_\alpha\}\,.
\end{aligned}$$

- $A = \parr$: we have $(\parr\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\})^\circ = \parr^\circ = \underline{\lambda}x.\underline{\Lambda}i_\beta.x = \parr^\circ\,\{\underline{\lambda}z.z/i_\alpha\}$.
- $A = \&$: we have $(\&\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\})^\circ = \&^\circ = \underline{\lambda}x.x \triangleleft \underline{\lambda}y.y = \&^\circ\,\{\underline{\lambda}z.z/i_\alpha\}$.
- $A = \phi;\psi$: we have

$$\begin{aligned}
((\phi;\psi)\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\})^\circ &= (\phi\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\}\,;\psi\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\})^\circ \\
&= \underline{\lambda}x.(\psi\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\})^\circ \triangleright ((\phi\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\})^\circ \triangleright x) \\
\text{(inductive hypothesis)} &= \underline{\lambda}x.(\psi^\circ\,\{\underline{\lambda}z.z/i_\alpha\}) \triangleright ((\phi^\circ\,\{\underline{\lambda}z.z/i_\alpha\}) \triangleright x) \\
&= (\underline{\lambda}x.\psi^\circ \triangleright (\phi^\circ \triangleright x))\,\{\underline{\lambda}z.z/i_\alpha\} \\
&= (\phi;\psi)^\circ\,\{\underline{\lambda}z.z/i_\alpha\}\,.
\end{aligned}$$

- $A = \mathbf{1}$: we have $(\mathbf{1}\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\})^\circ = \mathbf{1}^\circ = \underline{\lambda}x.x = \mathbf{1}^\circ\,\{\underline{\lambda}z.z/i_\alpha\}$.

For (iii), once again, the inductive steps where A is a term are immediate. Otherwise:

- $A = \sigma$: we have $(\sigma\,\{\phi;!\alpha/!\alpha\})^\circ = \sigma^\circ = (\underline{\lambda}x.x)\,\{(\underline{\lambda}z.i_\alpha \triangleright (\phi^\circ \triangleright z))/i_\alpha\} = \sigma^\circ\,\{(\underline{\lambda}z.i_\alpha \triangleright (\phi^\circ \triangleright z))/i_\alpha\}$.
- $A = !\alpha$: we have

$$\begin{aligned}
(!\alpha\,\{\phi;!\alpha/!\alpha\})^\circ &= (\phi;!\alpha)^\circ \\
&= \underline{\lambda}z.i_\alpha \triangleright (\phi^\circ \triangleright z) \\
&= i_\alpha\,\{\underline{\lambda}z.i_\alpha \triangleright (\phi^\circ \triangleright z)/i_\alpha\} \\
&= (!\alpha)^\circ\,\{\underline{\lambda}z.i_\alpha \triangleright (\phi^\circ \triangleright z)/i_\alpha\}\,.
\end{aligned}$$

– $A = \forall(\geq \phi)$: we have

$$\begin{aligned}
(\forall(\geq \phi)\,\{\phi; !\alpha/!\alpha\})^\circ &= (\forall(\geq \phi\,\{\phi; !\alpha/!\alpha\}))^\circ \\
&= \underline{\lambda} x.\underline{\not\lambda} i_\beta.x \lhd (\underline{\lambda} z.i_\beta \rhd ((\phi\,\{\phi; !\alpha/!\alpha\})^\circ \rhd z)) \\
(\text{ind. hyp.}) &= \underline{\lambda} x.\underline{\not\lambda} i_\beta.x \lhd (\underline{\lambda} z.i_\beta \rhd ((\phi^\circ\,\{\underline{\lambda} z.i_\alpha \rhd (\phi^\circ \rhd z)/i_\alpha\}) \rhd z)) \\
&= (\underline{\lambda} x.\underline{\not\lambda} i_\beta.x \lhd (\underline{\lambda} z.i_\beta \rhd (\phi^\circ \rhd z)))\,\{\underline{\lambda} z.i_\alpha \rhd (\phi^\circ \rhd z)/i_\alpha\} \\
&= (\forall(\geq \phi))^\circ\,\{\underline{\lambda} z.i_\alpha \rhd (\phi^\circ \rhd z)/i_\alpha\}\,.
\end{aligned}$$

– $A = \forall(\beta \geq)\phi$: we have (with $\beta \notin \mathrm{ftv}(\phi) \cup \{\alpha\}$)

$$\begin{aligned}
((\forall(\beta \geq)\phi)\,\{\phi; !\alpha/!\alpha\})^\circ &= (\forall(\beta \geq)\phi\,\{\phi; !\alpha/!\alpha\})^\circ \\
&= \underline{\lambda} z.i_\beta.(\phi\,\{\phi; !\alpha/!\alpha\})^\circ \rhd (x \lhd i_\beta) \\
(\text{ind. hyp.}) &= \underline{\lambda} z.i_\beta.(\phi^\circ\,\{\underline{\lambda} z.i_\alpha \rhd (\phi^\circ \rhd z)/i_\alpha\}) \rhd (x \lhd i_\beta) \\
&= (\underline{\lambda} z.i_\beta.\phi^\circ \rhd (x \lhd i_\beta))\,\{\underline{\lambda} z.i_\alpha \rhd (\phi^\circ \rhd z)/i_\alpha\} \\
&= (\forall(\beta \geq)\phi)^\circ\,\{\underline{\lambda} z.i_\alpha \rhd (\phi^\circ \rhd z)/i_\alpha\}\,.
\end{aligned}$$

– $A = \parr$: $(\parr\,\{\phi; !\alpha/!\alpha\})^\circ = \parr^\circ = \underline{\lambda} x.\underline{\not\lambda} i_\beta.x = \parr^\circ\,\{\underline{\lambda} z.i_\alpha \rhd (\phi^\circ \rhd z)/i_\alpha\}\,.$
– $A = \&$: we have $(\&\,\{\phi; !\alpha/!\alpha\})^\circ = \underline{\lambda} x.x \lhd \underline{\lambda} y.y = \&^\circ\,\{\underline{\lambda} z.i_\alpha \rhd (\phi^\circ \rhd z)/i_\alpha\}\,.$
– $A = \phi; \psi$: we have

$$\begin{aligned}
&((\phi;\psi)\ \{\phi; !\alpha/!\alpha\})^\circ \\
&\quad= (\phi\,\{\phi; !\alpha/!\alpha\}\,; \psi\,\{\phi; !\alpha/!\alpha\})^\circ \\
&\quad= \underline{\lambda} x.(\psi\,\{\phi; !\alpha/!\alpha\})^\circ \rhd ((\phi\,\{\phi; !\alpha/!\alpha\})^\circ \rhd x) \\
(\text{ind. hyp.})&\quad= \underline{\lambda} x.(\psi^\circ\,\{\underline{\lambda} z.i_\alpha \rhd (\phi^\circ \rhd z)/i_\alpha\}) \rhd ((\phi^\circ\,\{\underline{\lambda} z.i_\alpha \rhd (\phi^\circ \rhd z)/i_\alpha\}) \rhd x) \\
&\quad= (\underline{\lambda} x.\psi^\circ \rhd (\phi^\circ \rhd x))\,\{\underline{\lambda} z.i_\alpha \rhd (\phi^\circ \rhd z)/i_\alpha\} \\
&\quad= (\phi;\psi)^\circ\,\{\underline{\lambda} z.i_\alpha \rhd (\phi^\circ \rhd z)/i_\alpha\}\,.
\end{aligned}$$

– $A = \mathbf{1}$: we have $(\mathbf{1}\,\{\phi; !\alpha/!\alpha\})^\circ = \mathbf{1}^\circ = \underline{\lambda} x.x = \mathbf{1}^\circ\,\{\underline{\lambda} z.i_\alpha \rhd (\phi^\circ \rhd z)/i_\alpha\}$. □

Theorem 24 (Coercion calculus simulates $\mathsf{xML^F}$). *If $a \to_\beta b$ (resp. $a \to_\iota b$) in $\mathsf{xML^F}$, then $a^\circ \to_\beta b^\circ$ (resp. $a^\circ \xrightarrow{+}_{\mathsf{c}} b^\circ$) in coercion calculus.*

Proof. As the translation is contextual, it is sufficient to analyze each case of the reduction rules.

– $(\lambda(x:\tau)a)b \to_\beta a\,\{b/x\}$. We have $((\lambda(x:\tau)a)b)^\circ = (\lambda x.a^\circ)b^\circ$, β-reducing to $a^\circ\,\{b^\circ/x\}$, which is $(a\,\{b/x\})^\circ$ by Lemma 23(i).
– $a\mathbf{1} \to_\iota a$. We have $(a\mathbf{1})^\circ = \underline{\lambda} z.z \rhd a^\circ \to_{\mathsf{c}} z\,\{a^\circ/z\} = a^\circ$.
– $a(\phi;\psi) \to_\iota a\phi\psi$. We have $(a(\phi;\psi))^\circ = (\underline{\lambda} z.\psi^\circ \rhd (\phi^\circ \rhd z)) \rhd a^\circ \to_{\mathsf{c}} \psi^\circ \rhd (\phi^\circ \rhd a^\circ)$ which is equal to $(a\phi\psi)^\circ$.
– $a\parr \to_\iota \Lambda(\alpha \geq \bot)a$. Here we have $(a\parr)^\circ = (\underline{\lambda} x.\underline{\lambda} i_\alpha.x) \rhd a^\circ \to_{\mathsf{c}} \underline{\lambda} i_\alpha.a = (\Lambda(\alpha \geq \bot)a)^\circ$.
– $(\Lambda(\alpha \geq \tau)a)\& \to_\iota a\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\}$. Here, we have:

$$\begin{aligned}
((\Lambda(\alpha \geq \tau)a)\&)^\circ &= (\underline{\lambda} x.x \lhd \underline{\lambda} z.z) \rhd \underline{\not\lambda} i_\alpha.a^\circ \\
&\to_{\mathsf{c}} (\underline{\not\lambda} i_\alpha.a^\circ) \lhd \underline{\lambda} z.z \\
&\to_{\mathsf{c}} a^\circ\,\{\underline{\lambda} z.z/i_\alpha\} = (a\,\{\mathbf{1}/!\alpha\}\,\{\tau/\alpha\})^\circ, \text{ by Lemma 23(ii).}
\end{aligned}$$

– $(\Lambda(\alpha \geq \tau)a)(\forall(\alpha \geq)\phi) \to_\iota \Lambda(\alpha \geq \tau)a\phi$. We have:

$$\begin{aligned}\big((\Lambda(\alpha \geq \tau)a)(\forall(\alpha \geq)\phi)\big)^\circ &= (\underline{\lambda}x.\underline{\Lambda}i_\alpha.\phi^\circ \triangleright (x \triangleleft i_\alpha)) \triangleright (\underline{\Lambda}i_\alpha.a^\circ)\\ &\to_\mathsf{c} \underline{\Lambda}i_\alpha.\phi^\circ \triangleright ((\underline{\Lambda}i_\alpha.a^\circ) \triangleleft i_\alpha))\\ &\to_\mathsf{c} \underline{\Lambda}i_\alpha.\phi^\circ \triangleright a^\circ = (\Lambda(\alpha \geq \tau)a\phi)^\circ.\end{aligned}$$

– $(\Lambda(\alpha \geq \tau)a)(\forall(\geq \phi)) \to_\iota \Lambda(\alpha \geq \tau\phi)a\,\{\phi; !\alpha/!\alpha\}$. We have:

$$\begin{aligned}&\big((\Lambda(\alpha \geq \tau)a)(\forall(\geq \phi))\big)^\circ\\ &\quad= \big(\underline{\lambda}x.\underline{\Lambda}i_\alpha.x \triangleleft (\underline{\lambda}z.i_\alpha \triangleright (\phi^\circ \triangleright z))\big) \triangleright (\underline{\Lambda}i_\alpha.a^\circ)\\ &\quad\to_\mathsf{c} \underline{\Lambda}i_\alpha.(\underline{\Lambda}i_\alpha.a^\circ) \triangleleft (\underline{\lambda}z.i_\alpha \triangleright (\phi^\circ \triangleright z))\\ &\quad\to_\mathsf{c} \underline{\Lambda}i_\alpha.a^\circ\,\{(\underline{\lambda}z.i_\alpha \triangleright (\phi^\circ \triangleright z))/i_\alpha\}\\ &\quad= \underline{\Lambda}i_\alpha.(a\,\{\phi; !\alpha/!\alpha\})^\circ = (\Lambda(\alpha \geq \tau\phi)a\,\{\phi; !\alpha/!\alpha\})^\circ,\ \text{by Lemma 23(iii).} \quad \square\end{aligned}$$

Lemma 27.

(i) If $a^\circ \to_\beta b$ then $a \to_\beta c$ with $c^\circ = b$;
(ii) if $a^\circ \to_\mathsf{cv} b$ then $a \to_\iota c$ with $b \overset{=}{\to}_\mathsf{cv} c^\circ$.

$$\begin{tikzcd} a \arrow[r, dashed, "\beta"] \arrow[d, mapsto] & c \arrow[d, mapsto, dashed] \\ a^\circ \arrow[r, "\beta"] & b \end{tikzcd} \qquad \begin{tikzcd} a \arrow[rr, dashed, "\iota"] \arrow[d, mapsto] & & c \arrow[d, mapsto, dashed] \\ a^\circ \arrow[r, "\mathsf{cv}"] & b \arrow[r, dashed, "\mathsf{cv}="] & c^\circ \end{tikzcd}$$

Proof. By structural induction on a. Let us first settle point (i).

– $a = x$ ($a^\circ = x$): impossible.
– $a = \lambda(x : \tau)a_1$ ($a^\circ = \lambda x.a_1^\circ$), $\Lambda(\alpha \geq \tau)a_1$ ($a^\circ = \underline{\Lambda}i_\alpha.a_1^\circ$): the reduction takes necessarily place in a_1° and the inductive step is completed.
– $a = a_1\phi$ ($a^\circ = \phi^\circ \triangleright a_1^\circ$): we see that ϕ° is a coercion by Lemma 22(i) and is thus β-normal by Lemma 5, so the reduction takes place in a_1° and we proceed as above.
– $a = a_1a_2$: from $a^\circ = a_1^\circ a_2^\circ$ we can as above reduce to the case where a is the redex to be fired. We then have $a_1^\circ = \lambda x.a_3'$ and $b = a_3'\,\{a_2^\circ/x\}$, and necessarily $a_1 = \lambda(x : \tau)a_3$ with $a_3^\circ = a_3'$, so $a \to_\beta a_3\,\{a_2/x\}$ and $(a_3\,\{a_2/x\})^\circ = b$ by Lemma 23(i).

We now move to point (ii). We can exclude $a = x$, and the inductive steps are trivial for a equal to $\lambda(x : \tau)a_1$, a_1a_2 and $\Lambda(\alpha \geq \tau)a_1$, as the cv-reduction necessarily takes place in a strict subterm. It only remains the case $a = a_1\phi$, where $a^\circ = \phi^\circ \triangleright a_1^\circ$.

If a° is not the immediate redex of the reduction, then the latter must take place in a_1°, as ϕ° is typed as a coercion (Lemma 22(i)) and is thus cv-normal (Lemma 6). Inductive hypothesis then applies to a_1 and we are done.

Suppose therefore that $\phi^\circ \triangleright a_1^\circ$ is the redex being fired. The only way for a_1° to be a c-value is that either $a_1 = \lambda(x : \tau)a_3$ for any a_3, or $a_1 = \Lambda(\alpha \geq \tau)a_2$ (resp. $a_1 = a_2!\alpha$) with a_2° a c-value. First, we prove that $a_1\phi$ is necessarily a redex in xML^F. It would not be a redex only in the following cases.

– $\phi = \tau$: impossible as it requires a_1 to be of type $\bot$, which is excluded by all three alternatives for a_1.
– $\phi = !\beta$: this is likewise impossible as $\phi^\circ \triangleright a_1^\circ = i_\beta \triangleright a_1^\circ$ would not be a redex.

- a_1 not of the form $\Lambda(\alpha \geq \tau)a_2$ and $\phi = \&$, $\forall(\geq \psi)$ or $\forall(\alpha \geq)\psi$: excluding that a_1 starts with a Λ, we have $a_1 = \lambda(x : \tau)a_2$ or $a_1 = a_2!\alpha$. The type of a_1 would then be an arrow type or a type variable respectively, which are both incompatible with all the listed instantiations, which require a quantifier.

So there is c with $a_1\phi \to_\iota c$ obtained by firing $a_1\phi$ itself. Now take the steps $\phi^\circ \triangleright a_1^\circ \xrightarrow{*}_{\mathsf{c}} c^\circ$ simulating $a_1\phi \to_\iota c$, as shown in the proof of Theorem 24. We can then inspect such a proof and see that the first step always fires the redex $\phi^\circ \triangleright a_1^\circ$ (i.e. is the step we started with), which is then followed by at most one c-step, which is a cv one if a_1° is a c-value. ☐

B An Alternative Proof of Bisimulation

In this section we provide an alternative proof of Theorem 29, completely carried out within the xML^F system (given the SN result for xML^F). This proof is provided as a comparison to the one using $\mathsf{F_c}$. We first need this intermediate lemma, which is a version of Lemma 20 in xML^F.

Lemma 33. *If a is typable and ι-normal and $\lceil a \rceil = \lambda x.b$, then it is of one of the following forms, with c ι-normal:*

- *$a = \lambda(x : \tau)c$ with $\lceil c \rceil = b$;*
- *$a = \Lambda(\alpha \geq \tau)c$;*
- *$a = c!\alpha$.*

In particular if a is typed with some arrow type $\tau \to \sigma$, then $a = \lambda(x : \tau)c$.

Proof. By induction on a. As $\lceil a \rceil = \lambda x.b$ then a is neither an application nor a variable. Let us suppose that a is not of one of the above listed forms. The only remaining case is $a = a'\phi$ with a' ι-normal and $\phi \neq !\alpha$. By inductive hypothesis (as $\lceil a' \rceil = \lceil a \rceil = \lambda x.b$) we have that a' is one among $\lambda(x : \tau)c'$, $\Lambda(\alpha \geq \tau)c'$ and $c'!\alpha$, with c' ι-normal.

Now let us rule out all the cases for ϕ.

- $\phi = \sigma$: impossible as none of the three alternatives for a' is typable by $\bot$;
- $\phi = \mathbf{1}$, $\psi_1; \psi_2$ or $\parr$: impossible as $a'\phi$ would not be ι-normal;
- $\phi = \forall(\alpha \geq)\psi$, $\forall(\geq \psi)$ or $\&$: by typing a' must be $\Lambda(\alpha \geq \tau)c'$, as the other two alternatives would give an arrow and a variable type respectively, which is not compatible with these instantiations; however this is not possible as $a'\phi$ would form a ι-redex.

This concludes the proof. In case a has an arrow type $\tau \to \sigma$, the only compatible form is $a = \lambda(x : \tau)c$. ☐

Proof (Alternative proof of Theorem 29). The if part is immediate by verifying that $a \to_\iota^* a'$ implies $\lceil a \rceil = \lceil a' \rceil$, and $a' \to_\beta c$ implies $\lceil a' \rceil \to_\beta \lceil c \rceil$.

For the only if part, let a_0 be the ι-normal form of a (which exists as $\to_\iota$ is SN by Theorem 24). We have that $\lceil a_0 \rceil = \lceil a \rceil \to_\beta b$: if we prove that $a_0 \to_\beta c$ with $\lceil c \rceil = b$ we are done. Let us reason by induction on a_0.

- $a_0 = x$: impossible, as $\lceil a_0 \rceil = x$ is not reducible.
- $a_0 = \lambda(x : \tau)a_1$, $\Lambda(\alpha \geq \tau)a_1$ or $a_1\phi$: the reduction takes place in $\lceil a_1 \rceil$ and inductive hypothesis applies smoothly giving a β-reduction in a_1, and thus in a_0.
- $a_0 = a_1 a_2$: if the reduction takes place in $\lceil a_1 \rceil$ or $\lceil a_2 \rceil$ then the inductive hypothesis applies as above. Suppose then that $\lceil a_1 \rceil \lceil a_2 \rceil$ is itself the redex being fired, i.e. $\lceil a_1 \rceil = \lambda x.d$ and $b = d\,\{\lceil a_2 \rceil / x\}$. As a_1 is typed with some $\sigma \to \tau$ (in order to form the application) and $\lceil a_1 \rceil = \lambda x.d$, by Lemma 33 we have that $a_1 = \lambda(x : \sigma)a_3$ with $\lceil a_3 \rceil = d$, so $a_0 = (\lambda(x : \sigma)a_3)a_2 \to_\beta a_3\,\{a_2/x\}$ and $\lceil a_3\,\{a_2/x\} \rceil = d\,\{\lceil a_2 \rceil / x\} = b$. $\square$