Strong Normalization of $\mathsf{ML^F}$ via a Calculus of Coercions

Giulio Manzonetto[a,1], Paolo Tranquilli[b,2]

[a] *Intelligent Systems, Department of Computer Science, Radboud University, Nijmegen, The Netherlands*
[b] *LIP, CNRS UMR 5668, INRIA, ENS de Lyon, Université Claude Bernard Lyon 1, France*

Abstract

$\mathsf{ML^F}$ is a type system extending ML with first-class polymorphism as in system F. The main goal of the present paper is to show that $\mathsf{ML^F}$ enjoys strong normalization, i.e., it has no infinite reduction paths. The proof of this result is achieved in several steps. We first focus on $\mathsf{xML^F}$, the Church-style version of $\mathsf{ML^F}$, and show that it can be translated into a calculus of coercions: terms are mapped into terms and instantiations into coercions. This coercion calculus can be seen as a decorated version of system F, so that the simulation result entails strong normalization of $\mathsf{xML^F}$ through the same property of system F. We then transfer the result to all other versions of $\mathsf{ML^F}$ using the fact that they can be compiled into $\mathsf{xML^F}$ and showing there is a bisimulation between the two. We conclude by discussing what results and issues are encountered when using the candidates of reducibility approach to the same problem.

Keywords: $\mathsf{ML^F}$, $\mathsf{xML^F}$, calculus of coercions, 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 the program will be well-behaved at runtime. 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 the boring task of scattering the code with types to an automatic type reconstructor, part of the compiler. One of the most prominent

Email addresses: `g.manzonetto@cs.ru.nl` (Giulio Manzonetto), `paolo.tranquilli@ens-lyon.fr` (Paolo Tranquilli)

[1]Supported by NWO project CALMOC (612.000.936).
[2]Supported by ANR project COMPLICE (ANR-08-BLANC-0211-01).

examples of this approach is the functional programming language ML [1, 2, 3] and its dialects.

Polymorphism. In this context *type polymorphism* allows greater flexibility, as it makes 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 [4]) leads to undecidable type systems: no automatic reconstructor would be available [5]. For this reason ML employs 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 only such restricted polymorphism, even when a fully-polymorphic typing is known and could be provided by hand. 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.

Extending ML *with full polymorphism.* ML^F [6, 7] answers this call by providing a partial type annotation mechanism with an automatic type reconstructor. This extension allows to write system F programs, which is not possible in general in ML. Moreover it is a conservative extension: 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 $\bot$ as bound, where $\bot$ is morally equivalent to the usual $\forall\alpha.\alpha$. ML^F also uses a *rigid* quantifier $\forall(\alpha = \sigma)\tau$, fundamental for type inference but not for the semantics. Indeed $\forall(\alpha = \sigma)\tau$ can be regarded as being $\tau\,[\sigma/\alpha]$.

ML^F *and strong normalization.* One of the well-behaving properties 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 [4]. As already pointed out, system F is contained in ML^F. However it is not yet known, but it is conjectured [6], 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.

ML^F*'s variants.* ML^F comes in three versions with a varying degree of explicit typing. What we briefly described above and we might refer to as the "real deal" is in fact eML^F(following the nomenclature of [7]). In eML^F there are just enough type annotations to allow the automatic reconstruction of the missing ones, so that we may place it midway between the Curry and Church styles. The former is covered by the "implicit" version iML^F, where no type annotation whatsoever is present. Going the Church-style way we have a completely explicit version, $\mathsf{xML^F}$, studied in [8]. In $\mathsf{xML^F}$ 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 $\mathsf{eML^F}$.

With respect to $\mathsf{ML^F}$ the $\mathsf{xML^F}$ system is the main object of study in this work. Compared to Church-style system F, the type reduction $\to_\iota$ of $\mathsf{xML^F}$ is more complex, and may *a priori* cause non-termination, or block the reduction of a β-redex. The main difficulty lies in the non-trivial nature of the *type instance* relation $\sigma \leq \tau$. In $\mathsf{xML^F}$ for the sake of complete explicitness such relations are testified by syntactic entities called *instantiations* (see Figure 2). Given an instantiation $\phi : \sigma \leq \tau$ taking σ to τ and a term a of type σ the new term $a\phi$ will have the type τ. In fact ϕ plays the role of a type conversion, or in other words a *coercion*.

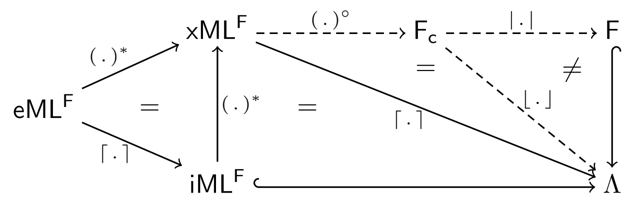
The coercion calculus. These type conversions have a non-trivial *type reduction* $\to_\iota$, as opposed to the easy type reduction of system F. Such a reduction may *a priori* introduce unexpected glitches in the system, such as introducing non-termination even if the β-reduction of the underlying term terminates, or on the contrary keeping a β-reduction of the underlying term from happening. To prove that none of this happens, rather than translating directly into system F we use an intermediate language abstracting the concept of coercion: the *coercion calculus* $\mathsf{F_c}$.

The delicate point in $\mathsf{xML^F}$ is that some of the 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 $\mathsf{xML^F}$, is in the form of a type system (Figure 6). 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 *simulation* with ordinary λ-calculus [9], 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, including strong normalization, to $\mathsf{xML^F}$ via a translation of the latter into the former (section 4). Its main idea is the same as for the one shown for $\mathsf{eML^F}$ in [10], where however no dynamic property was studied. We then produce a proof of SN for all versions of $\mathsf{ML^F}$, exploiting the fact $\mathsf{xML^F}$'s type erasure is a bisimulation. Such a result establishes that $\mathsf{xML^F}$ can indeed be used as an internal language for $\mathsf{eML^F}$, as the additional structure cannot block reductions of the intended program.

Candidates of reducibility. Before entering the details of the work, one may wonder whether the candidates of reducibility deliver the same result — indeed it was the first approach we tried. The *naïve* interpretation where type instantiation is mapped to inclusion of saturated sets (much like what has been done

The Big Picture



$(\,.\,)^*$ = compiling

$\lceil \,.\, \rceil$ = type erasure

$|\,.\,|$ = decoration erasure

$(\,.\,)^\circ$ = translation

$\lfloor \,.\, \rfloor$ = coercion erasure

Figure 1: Pre-existing relationships among the systems (solid arrows), plus our contribution (dashed arrows).

for $\mathsf{F}_{<:}$ [11]) works for the β-reduction of $\mathsf{xML^F}$, leaving outside the ι type reduction. As already explained, contrary to system F the latter is non-trivial, so its presence is another reason for embracing the system F translation approach. We will however give a presentation of the results using candidates of reducibility (or more precisely *saturated sets*) in section 6, and what glitches one encounters when dealing with the same approach with $\mathsf{eML^F}$ and $\mathsf{iML^F}$.

Contributions. The main results of the present paper are the proof of strong normalization for $\mathsf{xML^F}$ and its variants $\mathsf{eML^F}$ and $\mathsf{iML^F}$. Other contributions are the introduction of the coercion calculus $\mathsf{F_c}$, which is useful to better understand the coercion mechanism in the context of programming languages with first class polymorphism. We prove that $\mathsf{F_c}$ enjoys good properties like weakening, substitution, subject reduction and confluence. We show that $\mathsf{F_c}$ can be seen as a decoration of system F and that $\mathsf{F_c}$ reductions are translated into reductions in system F (simulation), therefore $\mathsf{F_c}$ enjoys the strong normalization property. From this result we then derive the strong normalization of $\mathsf{xML^F}$. The strong normalization of $\mathsf{eML^F}$ and $\mathsf{iML^F}$ is inferred from a (weak) bisimulation result which is enjoyed by $\mathsf{xML^F}$. Finally, we discuss how the strong normalization for $\mathsf{xML^F}$ can be also proved using the more standard technique of reducibility candidates (and why this approach is problematic for $\mathsf{eML^F}$ and $\mathsf{iML^F}$).

Outline. In Figure 1 we give a schematic representation of the interrelations among the various type systems that will be studied in the present paper. It is well known that the type erasure of $\mathsf{eML^F}$ terms gives $\mathsf{iML^F}$ terms [7] and that the two systems can be compiled into $\mathsf{xML^F}$ [8]. Obviously, we have that $\mathsf{iML^F}$ and system F are embeddable into the untyped λ-calculus, and the type erasure of $\mathsf{xML^F}$ terms gives ordinary λ-terms. This part of the picture was well-established in the literature.

We present $\mathsf{xML^F}$ in section 2 and the coercion calculus $\mathsf{F_c}$ in section 3. $\mathsf{F_c}$ is strongly normalizing as it can be seen as a decorated version of system F, where we denote by $|\,.\,|$ the *decoration erasure* (Definition 12). Moreover $\mathsf{F_c}$ enjoys the usual properties one expects of a type system, namely subject reduction. As coercions denote type conversions which morally have no operational meaning, a *coercion erasure* $\lfloor . \rfloor$ is given (Definition 18) extracting the actual semantics of a term. As shown in the diagram in Figure 1 the two mappings $\lceil . \rceil$ and $\lfloor . \rfloor$ to Λ are clearly different.

We then move to one of the main contributions of the paper by defining in section 4 a translation $(\,.\,)^\circ$ from $\mathsf{xML^F}$ to $\mathsf{F_c}$ (Figure 9). In this way we prove that $\mathsf{xML^F}$ is strongly normalizing: suppose indeed that there is an infinite reduction chain in $\mathsf{xML^F}$, then it is simulated via the translation $(\,.\,)^\circ$ in $\mathsf{F_c}$, which is impossible. However $\mathsf{F_c}$ does not enjoy bisimulation.

To entail the same result for $\mathsf{eML^F}$ and $\mathsf{iML^F}$ we need to be sure that any infinite reduction in one of the two systems can be lifted to an infinite one in $\mathsf{xML^F}$. This is achieved in section 5 by proving that the type erasure $\lceil . \rceil$ from $\mathsf{xML^F}$ to the λ-calculus Λ (Definition 3) is in fact a (weak) bisimulation (Theorem 33).

Finally in section 6 we define a candidates of reducibility interpretation for $\mathsf{xML^F}$ types, implying SN of $\lceil a \rceil$ for $\mathsf{xML^F}$ terms a, but failing to directly provide the full result.

Notations and basic definitions. 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. A reduction $\to$ is *strongly normalizing* if there is no infinite chain $a_i \to a_{i+1}$; it is *confluent* if $\xleftarrow{*}\xrightarrow{*} \subseteq \xrightarrow{*}\xleftarrow{*}$. In confluence diagrams, solid arrows denote reductions one starts with, while dashed arrows are the entailed ones.

2. A Short Presentation of $\mathsf{xML^F}$

Currently, $\mathsf{ML^F}$ comes in a Curry-style version $\mathsf{iML^F}$, where no annotation is provided, and a type-inference version $\mathsf{eML^F}$ requiring partial annotations, though a large amount of type information is automatically inferred. A truly Church-style version of $\mathsf{ML^F}$, called $\mathsf{xML^F}$, has been recently introduced in [8] and will be our main object of study in this paper. However, in section 5, we will draw conclusions for $\mathsf{iML^F}$ and $\mathsf{eML^F}$ too.

We warn the reader that we will only present the definitions we need, while we refer to [8] for an in-depth discussion on $\mathsf{xML^F}$. Concerning the presentation of $\mathsf{iML^F}$ and $\mathsf{eML^F}$ we refer to [12, 13].

2.1. Syntax

All the syntactic definitions of $\mathsf{xML^F}$ can be found in Figure 2. To be consistent with the existing literature we use the same notations of [8], but we warn the reader that the instantiations $\&, \parr$ and $!\alpha$ have no connection whatsoever with the "par", "with" and "promotion" connectives of linear logic.

We assume fixed a countable set of **type variables** denoted by $\alpha, \beta, \ldots$

$\alpha, \beta, \ldots$		(type variables)
σ, τ	$::= \alpha \mid \sigma \to \tau \mid \bot \mid \forall(\alpha \geq \sigma)\tau$	(types)
ϕ, ψ	$::= \tau \mid \phi; \psi \mid \mathbf{1} \mid \& \mid \invamp \mid !\alpha \mid \forall(\geq \phi) \mid \forall(\alpha \geq)\phi$	(instantiations)
$x, y, z, \ldots$		(variables)
a, b, c	$::= x \mid \lambda(x : \tau)a \mid ab \mid \Lambda(\alpha \geq \tau)a \mid a\phi \mid \mathtt{let}\ x = a\ \mathtt{in}\ b$	(terms)
A, B	$::= a \mid \phi$	(expressions)
Γ	$::= \emptyset \mid \Gamma, \alpha \geq \tau \mid \Gamma, x : \tau$	(environments)

Figure 2: Syntactic definitions of $\mathsf{xML^F}$.

Types include type variables and arrow types, as usual. Here types also contain a bottom type $\bot$ corresponding to system F's type $\forall\alpha.\alpha$ and 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. We write $\mathrm{ftv}(\tau)$ for the set of type variables appearing free in a type τ.

An **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. We write $\mathrm{ftv}(\phi)$ for the set of free type variables of ϕ.

Terms of $\mathsf{xML^F}$ extend the ordinary λ-terms with a constructor `let`, type instantiation and type application. Type instantiation $a\phi$ generalizes system F type application. Type abstractions are extended with an instance bound τ, written $\Lambda(\alpha \geq \tau)a$. The type variable α is bounded in a, but free in τ. We write $\mathrm{fv}(a)$ (resp. $\mathrm{ftv}(a)$) for the set of free term (resp. type) variables of a.

Expressions can be either terms or instantiations. They are not essential for the calculus, but will be used to state results holding for both syntactic categories in a more elegant and compact way.

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

2.2. Type System

Typing rules of $\mathsf{xML^F}$ are provided in Figure 3. **Typing judgments** are of the form $\Gamma \vdash a : \tau$, where a is an $\mathsf{xML^F}$ term, Γ a (well-formed) environment and τ a type. Especially focus on type abstraction and type instantiation that are the biggest novelties with respect to system F. Type abstraction $\Lambda(\alpha \geq \tau)a$ extends the environment Γ with the type variable α bounded by τ. Notice that the typing of a type instantiation $a\phi$ is similar to the typing of a coercion, as it just requires the instantiation ϕ to transform the type of a to the type of the result. This analogy will be formally developed in section 4. The `let`-binding

Instantiation rules

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

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

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

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

Typing rules

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

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

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

Type instantiation

$$\tau(!\alpha) := \alpha, \qquad \bot\tau := \tau, \qquad \tau\mathbf{1} := \tau, \qquad \tau(\phi;\psi) := (\tau\phi)\psi,$$
$$\tau\parr := \forall(\alpha \geq \bot)\tau, \quad \alpha \notin \mathrm{ftv}(\tau), \qquad (\forall(\alpha \geq \sigma)\tau)(\forall(\geq \phi)) := \forall(\alpha \geq \sigma\phi)\tau,$$
$$(\forall(\alpha \geq \sigma)\tau)\& := \tau\,[\sigma/\alpha], \qquad (\forall(\alpha \geq \sigma)\tau)(\forall(\alpha \geq)\phi) := \forall(\alpha \geq \sigma)(\tau\phi).$$

Figure 3: The typing rules of $\mathsf{xML^F}$.

$\texttt{let}\ x = a\ \texttt{in}\ b$ is morally equivalent to the immediate application $(\lambda(x : \tau)b)a$ except that in the `let` the variable x does not require type annotation. We will soon forget about the `let` (see Convention 2, below) as it is unnecessary for our study.

Type instance judgments have the shape $\Gamma \vdash \phi : \sigma \leq \tau$ stating that in the environment Γ the instantiation ϕ maps the type σ into the type τ.

The bottom instantiation states that every type τ is an instance of $\bot$, independently of the environment. The abstract instantiation $!\alpha$ is applicable in an environment containing $\alpha \geq \tau$ and abstracts the bound τ of α as the type variable α. The inside instantiation $\forall(\geq \phi)$ applies ϕ to the bound σ of a flexible quantification $\forall(\beta \geq \sigma)\tau$. Conversely, the under instantiation $\forall(\alpha \geq)\phi$ applies ϕ to the type τ under the quantification. The quantifier introduction $\parr$ introduces a fresh trivial quantification $\forall(\alpha \geq \bot)$. *Vice versa*, the quantifier elimination $\&$ eliminates the bound of a type of the form $\forall(\alpha \geq \tau)\sigma$ by substituting τ for α in σ. The composition $\phi; \psi$ provides a witness of the transitivity of type instance, while the identity instantiation $\mathbf{1}$ of reflexivity.

Instantiations give a computational account of the instance relation holding between types and can be better understood looking at their dynamical

$$
\begin{aligned}
(\lambda(x:\tau)a)b &\to_\beta a\,[b/x]\\
\texttt{let}\ x = b\ \texttt{in}\ a &\to_\beta a\,[b/x]\\
a\mathbf{1} &\to_\iota a\\
a(\phi;\psi) &\to_\iota (a\phi)\psi\\
a\text{⅋} &\to_\iota \Lambda(\alpha \geq \bot)a,\ \alpha \notin \mathrm{ftv}(a)\\
(\Lambda(\alpha \geq \tau)a)\& &\to_\iota a\,[\mathbf{1}/!\alpha]\,[\tau/\alpha]\\
(\Lambda(\alpha \geq \tau)a)(\forall(\alpha \geq)\phi) &\to_\iota \Lambda(\alpha \geq \tau)(a\phi)\\
(\Lambda(\alpha \geq \tau)a)(\forall(\geq \phi)) &\to_\iota \Lambda(\alpha \geq \tau\phi)a\,[\phi; !\alpha/!\alpha]
\end{aligned}
$$

Figure 4: Reduction rules of $\mathsf{xML^F}$.

semantics presented in the next subsection.

In $\mathsf{iML^F}$ flexible quantification allows us to recover the property of *principal types* that was lost in system F. This phenomenon can be observed also in $\mathsf{xML^F}$, e.g. in the following paradigmatic example. Let `choice` be a system F program of type $\forall\alpha.\alpha \to \alpha \to \alpha$, e.g. $\lambda x.\lambda y.x$ and id be the identity program $\lambda x.x$ of type $\forall\alpha.\alpha \to \alpha$. The application of `choice` to id has several types in system F that are incompatible: for instance it can be typed both with $(\forall\beta.\beta \to \beta) \to (\forall\beta.\beta \to \beta)$ and with $\forall\gamma.(\gamma \to \gamma) \to (\gamma \to \gamma)$.

In $\mathsf{xML^F}$ we write the polymorphic identity $\mathrm{id} = \Lambda(\alpha \geq \bot)\lambda(x : \alpha)x$ of type $\tau_{\mathrm{id}} = \forall(\alpha \geq \bot)(\alpha \to \alpha)$. A possible implementation of the aforementioned function `choice` is $\Lambda(\beta \geq \bot)\lambda(x : \beta)\lambda(y : \beta)x$ of type $\forall(\beta \geq \bot)\beta \to \beta \to \beta$. The application of `choice` to id can be defined as the program

$$\texttt{choice\_id} = \Lambda(\beta \geq \tau_{\mathrm{id}})\texttt{choice}\langle\beta\rangle(\mathrm{id}(!\beta)),\ \text{where } \langle\beta\rangle = \forall(\geq \beta); \&.$$

We can give weaker types to `choice_id` by type instantiation; for instance we can recover the two system F types mentioned above. Indeed the term $\texttt{choice\_id}\&$ has type $(\forall(\beta \geq \bot)\beta \to \beta) \to (\forall(\beta \geq \bot)\beta \to \beta)$, while the term $\texttt{choice\_id}(\text{⅋}; \forall(\gamma \geq)(\forall(\geq \langle\gamma\rangle); \&))$ has type $\forall(\gamma \geq \bot)(\gamma \to \gamma) \to (\gamma \to \gamma)$.

2.3. Operational Semantics

One of the main technical aspects of $\mathsf{xML^F}$ is presenting how type instantiations evolve during reduction. $\mathsf{xML^F}$'s reduction rules are presented in Figure 4. They are divided into $\to_\beta$ (regular β-reductions) and $\to_\iota$, reducing instantiations. We allow reductions to occur in any context, including under λ-abstractions. Note that the last 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.

We recall, from [8, Sec. 2.1], that both $\to_\beta$ and $\to_\iota$ enjoy subject reduction.

Lemma 1 (Subject reduction)**.** *Let a be an $\mathsf{xML^F}$ term.*

(i) $\Gamma \vdash a : \sigma$ *and* $a \to_\beta b$ *entail* $\Gamma \vdash b : \sigma$*,*

(ii) $\Gamma \vdash a : \sigma$ *and* $a \to_\iota b$ *entail* $\Gamma \vdash b : \sigma$*.*

Hereafter, we will adopt the following convention.

Convention 2. Here we presented the original syntax of $\mathsf{xML^F}$ which also contains the `let` construct. However this instruction has been added mainly to accommodate $\mathsf{eML^F}$'s type reconstructor. Hence in the whole paper we can suppose that in all $\mathsf{xML^F}$ terms every $\texttt{let}\ x = a\ \texttt{in}\ b$ has been replaced by $(\lambda(x : \sigma)b)a$, with σ the correct type of a.

We end the section by defining the type erasure of an $\mathsf{xML^F}$ term, which erases all type and instantiation annotations, mapping a to an ordinary λ-term.

Definition 3. The **type erasure** $\lceil a \rceil$ of an $\mathsf{xML^F}$ term a is defined by:

$$\lceil x \rceil := x, \quad \lceil \lambda(x : \tau)a \rceil := \lambda x.\lceil a \rceil, \quad \lceil ab \rceil := \lceil a \rceil \lceil b \rceil,$$
$$\lceil \Lambda(\alpha \geq \sigma)a \rceil := \lceil a \rceil, \quad \lceil a\phi \rceil := \lceil a \rceil.$$

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 3.5) 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 meta-theory of coercions, more liberal choices and constructs are needed, as discussed at page 29. The system $\mathsf{F_c}$ is a general language for coercions, as for example the one presented in [14] or more recently in [15]. Calculi for coercions have two main points of interest. On the one side they provide a meta-theory for calculi where type conversions are left implicit, allowing for an easier reasoning on them. On the other side, they could be useful as intermediate languages, allowing the compilation or execution of languages with type conversions to retain some form of type information.

A note on $\mathsf{F_c}$ *and* DILL. The type system we will present can be said to be a subsystem of lambda calculus typed with *dual intuitionistic linear logic* derivations (DILL, [16]). Such a system, built on top of linear logic [17], is characterized by having judgments of the form $\Gamma; L \vdash A$, where the context is split in a linear part L whose assumptions may be used just once and a regular, non-linear part Γ. Here the linear context and the linear arrow $\multimap$ will capture the linearity aspect of coercions: they neither erase nor duplicate their arguments.

The language presented in [16] 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. So technically speaking employing DILL as a type system for ordinary λ-terms leads to another system (which we might call F_ℓ) using types rather than terms to strictly differentiate between linear and regular constructs. This system is known in *folklore*[3] but, as far as we know, it has never been thoroughly presented

[3] As an example we might cite [18], where a fragment of F_ℓ is used to characterize poly-time functions.

$$
\begin{array}{lr}
\alpha, \beta, \ldots & \text{(type variables)} \\
\sigma, \tau ::= \alpha \mid \sigma \to \tau \mid \kappa \to \tau \mid \forall\alpha.\tau & \text{(types)} \\
\kappa \quad ::= \sigma \multimap \tau & \text{(coercion types)} \\
\zeta \quad ::= \tau \mid \kappa & \text{(type expressions)} \\
x, y, z, \ldots & \text{(variables)} \\
a, b ::= x \mid \lambda x.a \mid \underline{\lambda} x.a \mid \underline{\measuredangle} x.a \mid ab \mid a \triangleright b \mid a \triangleleft b & \text{(terms)} \\
u, v ::= \lambda x.a \mid \underline{\measuredangle} x.u \mid x \triangleright u & \text{(}\mathtt{c}\text{-values)} \\
\Gamma \quad ::= \emptyset \mid x : \tau, \Gamma \mid x : \sigma \multimap \alpha, \Gamma & \text{(regular environments)} \\
L \quad ::= \emptyset \mid z : \tau & \text{(linear environments)} \\
\Gamma; L & \text{(environments)} \\
\Gamma; \vdash_{\mathtt{t}} a : \sigma & \text{(term judgements)} \\
\Gamma; \vdash_{\mathtt{c}} a : \kappa & \text{(coercion judgements)} \\
\Gamma; z : \tau \vdash_{\ell} a : \sigma & \text{(linear judgements)} \\
\vdash_{\mathtt{xy}},\ \mathtt{x}, \mathtt{y} \in \{\, \mathtt{t}, \mathtt{c}, \ell \,\} \text{ stands for } \vdash_{\mathtt{x}} \text{ or } \vdash_{\mathtt{y}}. &
\end{array}
$$

Figure 5: Syntactic definitions of coercion calculus.

in the literature.

3.1. Syntax

The syntactic categories of (Curry-style) coercion calculus are presented in Figure 5.

In **types** the difference from usual system F types lies in the presence of a new arrow for coercions, denoted by the lollipop $\multimap$. As already explained above, contrary to $\mathsf{xML}^{\mathsf{F}}$'s notation here the use of the linear logic symbol is pertinent. These **coercion types** $\sigma \multimap \tau$ will type conversions from the type σ to the type τ and are allowed to appear in regular types only on the left of an arrow. These in fact leads to three distinguishable arrow types: regular with regular type on the left, regular with coercion type on the left and finally the coercion arrow. For type polymorphism $\forall\alpha.\tau$ we employ a different typesetting convention with respect to $\mathsf{xML}^{\mathsf{F}}$'s types for the sake of clarity. **Type expressions** denote both sorts of types.

We reflect the three different kinds of arrow types in **terms** with three different abstraction/application pairs. These are to be intended as mere decorations of the usual pair, used both to distinguish regular reduction from coercion one (subsection 3.3) and to define coercion erasure (subsection 3.6) directly on terms without regarding their type derivation. The three different pairs of abstraction/application are

- the regular one with $\lambda x.a$ and ab, where no coercion is involved;
- the **linear abstraction** and **application** $\underline{\lambda} x.a$ and $a \triangleright b$: the former builds a coercion and the latter applies the coercion a to the term b;
- the **coercion abstraction** and **application** $\underline{\measuredangle} x.a$ and $a \triangleleft b$: the former expects a coercion to be passed to it, which is achieved by the latter where the coercion b is passed to a.

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

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

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

$$
\frac{\Gamma; L \vdash_{\mathtt{t}\ell} a : \sigma \qquad \alpha \notin \mathrm{ftv}(\Gamma; L)}{\Gamma; L \vdash_{\mathtt{t}\ell} a : \forall \alpha.\sigma}\,\textsc{Gen}
\qquad
\frac{\Gamma; L \vdash_{\mathtt{t}\ell} a : \forall \alpha.\sigma}{\Gamma; L \vdash_{\mathtt{t}\ell} a : \sigma\,[\tau/\alpha]}\,\textsc{Inst}
$$

Figure 6: Typing rules of coercion calculus.

Notice that in applications the side of the triangle indicates where the coercion is.

Here we moreover introduce a special subclass of terms which we call **c-values**. Essentially they are regular abstractions wrapped in the "blocking" coercion operations: coercion abstraction and linear application with a variable in coercion position. Its role will be made more clear when we will discuss $\mathsf{F_c}$'s reductions.

Environments are of shape $\Gamma; L$, where Γ is a map from term variables to type expressions (a **regular environment**), and L is the **linear environment**, containing (contrary to DILL) *at most* one assignment.

3.2. Typing Rules

In $\mathsf{F_c}$ typing judgments are of the general form $\Gamma; L \vdash M : \zeta$. However the shape of the environment L (which can be either empty or containing one assignment) and of the type ζ (which can be regular or a coercion one) gives four different general combinations. Of these only three will be allowed by the rules:

- no linear assignment and a regular type gives rise to a **term judgment**, i.e. the typing of a regular term, marked by $\vdash_{\mathtt{t}}$;
- no linear assignment and a coercion type is a **coercion judgment**, which is marked by $\vdash_{\mathtt{c}}$;
- a linear assignment and a regular type is a **linear judgment**, and denotes in fact the building in progress of a coercion, marked by $\vdash_{\ell}$.

So in fact the subscripts of $\vdash$ are there just as an aid to readability, as they can be completely recovered from the shape of the judgment.

The typing rules making up $\mathsf{F_c}$ are presented in Figure 6. With the rules at hand we can finally specify what exactly a coercion is in our framework.

Definition 4 (Coercion and regular terms)**.** An $\mathsf{F_c}$ term a is:

- a **coercion** if $\Gamma; \vdash_{\mathtt{c}} a : \sigma \multimap \tau$,

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

Figure 7: Reduction rules of coercion calculus.

- **regular** if $\Gamma; \vdash_{\mathsf{t}} a : \sigma$.

There are three main ideas behind the design of $\mathsf{F_c}$'s typing rules.

- Regular operations (i.e. not marked as coercion or linear ones) are allowed only while building a regular term and not in coercions, so Abs and App are only on $\vdash_{\mathsf{t}}$ judgments.
- For the context L to be linear means that in the rules with two premises (namely LApp and CApp), it will be just on one side. As the linear variable stands for the term to be coerced, it will not be on the side of the coercion in the two aforementioned rules.
- The system is tailored for the needs of $\mathsf{xML^F}$, so some restrictions have been made: for example coercions cannot be themselves coerced and are not polymorphic.

Discussing the rules some more in detail, we see that Ax is the usual axiom which can also introduce coercion variables, while Lax is its linear version used to start building a coercion. LAbs is the only other rule (with Ax) introducing coercions, and together with LApp they type the linear abstraction-application pair, available both for terms and for coercions under construction. The third abstraction-application pair is left to the CAbs and CApp rules.

3.3. Operational Semantics

Regarding reduction rules there is in fact not much to say as the different kinds of abstraction/application pairs are decorations of the usual one and as such share its reduction rules. This is shown in Figure 7, and as usual the rules are to be intended closed by context. The only detail to observe is that we distinguish regular β-reductions (denoted by $\to_\beta$) from the **coercion reductions** (denoted by $\to_{\mathsf{c}}$) which as the name suggests concern the coercion part of the terms.

3.4. Some Basic Properties of $\mathsf{F_c}$

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

Remark 5. 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.

Lemma 6. *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.*

Proof. Let us here call *strictly 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 proper subterm of a (i.e. no subterm of a') is strictly regular.

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

- a is not strictly 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). □

Following are basic properties of type systems. Note that though there are two substitution results (points (i), (ii) of Lemma 8 below) to accommodate the two types of environment, no weakening property is available to add the linear assignment.

Lemma 7 (Weakening). *We have that* $\Gamma; L \vdash_{\mathtt{tc}\ell} a : \zeta$ *and* $x \notin \mathrm{dom}(\Gamma; L)$ *entail* $\Gamma, x : \zeta'; L \vdash_{\mathtt{tc}\ell} a : \zeta$*;*

Proof. Trivial induction on the size of the derivation. As usual, one may have to change the bound variable in the Gen rule. □

Lemma 8 (Substitution). *We have the following:*

(i) $\Gamma; \vdash_{\mathtt{tc}} a : \zeta'$ *and* $\Gamma, x : \zeta'; L \vdash_{\mathtt{tc}\ell} b : \zeta$ *entail* $\Gamma; L \vdash_{\mathtt{tc}\ell} b\,[a/x] : \zeta$*;*

(ii) $\Gamma; L \vdash_{\mathtt{t}\ell} a : \sigma$ *and* $\Gamma; x : \sigma \vdash_{\ell} b : \zeta$ *entail* $\Gamma; L \vdash_{\mathtt{t}\ell} b\,[a/x] : \zeta$.

Proof. Both substitution results are obtained by induction on the derivation for b, by cases on its last rule.

- Ax: for (i), 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 (i) $L = z : \sigma$ and $b = z \neq x$, so $\Gamma; z : \sigma \vdash_{\ell} z = z\,[a/x] : \sigma$ and we are done; for (ii) necessarily $b = x$, $\zeta = \sigma$ and $b\,[a/x] = a$ and we are done.
- Abs, App and LAbs: trivial application of inductive hypothesis for (i), while it does not apply for (ii) as the judgment for b cannot be a linear one.
- CAbs, Gen and Inst: for these unary rules both (i) and (ii) are trivial.
- CApp and LApp: for (i) the substitution distributes as usual; for (ii) 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 next standard lemma is used in some of the following results.

Lemma 9. *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 the 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:

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

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. $\square$

We now show that the coercion calculus satisfies both subject reduction and confluence.

Proposition 10 (Subject reduction)**.** *If $\Gamma; L \vdash_{\mathtt{t}\ell\mathtt{c}} a : \zeta$ and $a \to_{\beta\mathtt{c}} b$ then $\Gamma; L \vdash_{\mathtt{t}\ell\mathtt{c}} b : \zeta$.*

Proof. By Lemma 9 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 $(\lambda x.c)d$, $(\underline{\measuredangle} x.c) \triangleleft d$ or $(\underline{\lambda} x.c) \triangleright d$. 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 $\Gamma, x : \sigma; \vdash_{\mathtt{t}} c : \tau$, $\Gamma; \vdash_{\mathtt{t}} d : \sigma$ and Lemma 8(i) settles the case;
- $(\underline{\measuredangle} x.c) \triangleleft d \to_{\mathtt{c}} c\,[d/x]$: 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 8(i) entails the result;
- $(\underline{\lambda} x.c) \triangleright d \to_{\mathtt{c}} c\,[d/x]$: 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 8(ii) that applies. $\square$

Syntactic categories

$$\begin{array}{ll}
\alpha, \beta, \dots & \text{(type variables)} \\
\sigma, \tau ::= \alpha \mid \sigma \to \tau \mid \forall \alpha.\tau & \text{(types)} \\
x, y, z, \dots & \text{(variables)} \\
a, b ::= x \mid \lambda x.a \mid ab \mid & \text{(terms)} \\
\Gamma \quad ::= \emptyset \mid \Gamma, x : \tau & \text{(environments)}
\end{array}$$

Typing rules

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

$$\frac{\Gamma \vdash_{\mathsf{F}} a : \sigma \qquad \alpha \notin \mathrm{ftv}(\Gamma)}{\Gamma \vdash_{\mathsf{F}} a : \forall \alpha.\sigma}\textsc{Gen} \qquad \frac{\Gamma \vdash_{\mathsf{F}} a : \forall \alpha.\sigma}{\Gamma \vdash_{\mathsf{F}} a : \sigma\,[\tau/\alpha]}\textsc{Inst}$$

Figure 8: Syntax and typing rules of Curry-style system F.

Proposition 11 (Confluence)**.** *All of $\to_\beta$, $\to_{\mathsf{c}}$ and $\to_{\beta\mathsf{c}}$ are confluent.*

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

3.5. 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 [4], which for the sake of completeness is briefly recalled in Figure 8.

System F can be recovered by collapsing the extraneous constructs $\multimap$, $\underline{\lambda}$, $\underline{\measuredangle}$, $\triangleleft$ and $\triangleright$ to their regular counterpart. Notably this will lead to a strong normalization result.

Definition 12. The **decoration erasure** of $\mathsf{F_c}$ types and terms is defined by:

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

Lemma 14 ensures that the decoration erasure is sound with respect to typability. We just need the standard weakening lemma for system F, which we state for completeness.

Lemma 13. *If $\Gamma \vdash_{\mathsf{F}} a : \sigma$ and $x \notin \mathrm{dom}(\Gamma)$ then $\Gamma, x : \tau \vdash_{\mathsf{F}} a : \sigma$.*

Lemma 14. *Let a be an $\mathsf{F_c}$ term. If $\Gamma; L \vdash_{\mathtt{tc}\ell} a : \zeta$ 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 Lemma 13 will have to be applied to add the $z : |\tau|$ coming from the linear environment which will be missing in one of the two branches. □

It is now immediate to see how decoration erasure agrees with substitution and thus reduction.

Lemma 15. *Given an $\mathsf{F_c}$ term a we have $|a\,[b/x]\,| = |a|\,[|b|/x]$.*

Proof. Trivial by induction on the term. $\square$

Lemma 16. *Let a be typable in $\mathsf{F_c}$. If $a \to_{\beta\mathsf{c}} b$ then $|a| \to |b|$. Vice versa $|a| \to c$ implies $c = |b|$ with $a \to_{\beta\mathsf{c}} b$.*

Proof. The first claim is immediate from Lemma 15. The converse needs typability of a: 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 ($(\lambda x.b_1)b_2$, $(\underline{\lambda}x.b_1)b_2$, $(\lambda x.b_1) \lhd b_2$, etc.). However as a is typable only the three matching combinations are possible, giving rise to the three possible redexes in the coercion calculus. $\square$

As an easy consequence we get that $\mathsf{F_c}$ is strongly normalizing.

Corollary 17 (Termination)**.** *The coercion calculus is strongly normalizing.*

Proof. Immediate by Lemmas 14 and 16, using the strong normalization property of system F [4, Sec. 14.3]. $\square$

3.6. 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. We will then establish how reductions in coercion calculus can be stripped of the coercion reductions to recover actual β-reductions in the semantics.

Definition 18. The **coercion erasure** is a map from $\mathsf{F_c}$ terms to regular λ-calculus defined by:

$$\begin{gathered}\lfloor x \rfloor := x, \quad \lfloor \lambda x.a \rfloor := \lambda x.\lfloor a \rfloor, \quad \lfloor ab \rfloor := \lfloor a \rfloor \lfloor b \rfloor,\\ \lfloor \underline{\Lambda} x.a \rfloor := \lfloor a \rfloor, \quad \lfloor a \lhd b \rfloor := \lfloor a \rfloor, \quad \lfloor a \rhd b \rfloor := \lfloor b \rfloor.\end{gathered}$$

Notice that it is undefined on $\underline{\lambda}x.a$ terms, as we will not apply it to coercions.

Lemma 19.

(i) If $\Gamma, x : \kappa; L \vdash_{\mathtt{t}\ell} a : \sigma$ (i.e. x is a coercion variable) 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 $a = z$ 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. $\square$

Notice that property (i) above entails that $\lfloor\,.\,\rfloor$ is well-defined with respect to α-equivalence on regular, typed terms: given a term $\underline{\measuredangle}x.a$ issued from a coercion abstraction, $\lfloor \underline{\measuredangle}x.a \rfloor = \lfloor a \rfloor$ is independent from 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 5), with $\lfloor a' \rfloor = x$. Apart when they are variables, coercions are essentially identities.

We turn back to study the properties of the coercion erasure, firstly by stating a fundamental and easy result on its interaction with substitution.

Lemma 20. *For $\mathsf{F_c}$ terms a and b we have that $\lfloor a\,[b/x] \rfloor = \lfloor a \rfloor\,[\lfloor b \rfloor/x]$, when both sides are defined*[4].

Proof. Immediate induction. $\square$

The following result employs the linearity constraint in a crucial way: reductions in linear position can be neither erased nor duplicated.

Lemma 21. *If $\Gamma; x : \tau \vdash_\ell a : \sigma$ and $b \to_\beta c$, then $a\,[b/x] \to_\beta a\,[c/x]$.*

Proof. The proof is an easy induction on the derivation.

We proceed 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. $\square$

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)).

Proposition 22. *Suppose that a is an $\mathsf{F_c}$ 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 $\lfloor a \rfloor = \lfloor b \rfloor$.

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

[4] We regard the right-hand side to be defined even if $\lfloor b \rfloor$ is not defined but $x \notin \mathrm{fv}(\lfloor a \rfloor)$, in which case we simply take $\lfloor a \rfloor$.

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 steps may be needed to close the diagram).

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 $\Gamma; \vdash_{\mathsf{t}} e : \sigma$. 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 21 we have that $b_1 \to_\beta d\,[e'/x]$ and we are done.

If firing $a = (\underline{\measuredangle} x.d) \triangleleft e$ we have that e is a coercion, which cannot contain any β-redex, so we have $d \to_\beta d'$ and $b_2 = (\underline{\measuredangle} x.d') \triangleleft e$. We easily get $b_2 \to_{\mathsf{c}} d'\,[e/x] \leftarrow_\beta d\,[e/x] = b_1$.

(ii) By induction and β-normality of coercions we can reduce to the case where $a = (\lambda x.c)d$. By Lemma 20, 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 $a = (\underline{\measuredangle} x.c) \triangleleft d$ (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 19(i) and 20 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 19(ii) and 20 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. ∎

3.7. The absence of bisimulation

As shown above, $\mathsf{F_c}$ enjoys good properties, and has a straightforward interpretation of what coercions are. However adding the ability to abstract over coercions (and thus having coercion variables) brings in a problem: coercion variables can block regular β-reduction. In other words, while coercion erasure grants simulation, it is not a bisimulation, i.e. there can be reductions in the coercion erasure that do not lift to reductions in $\mathsf{F_c}$.

Fact 23. There is a normal $\mathsf{F_c}$ term whose coercion erasure is not normal: take for example $\underline{\measuredangle} x.(x \triangleright I)I$, with $I = \lambda z.z$, whose coercion erasure is II.

Indeed, the situation is even worse.

Fact 24. The coercion erasure on $\mathsf{F_c}$ is surjective on the whole untyped λ-calculus.

Proof. Take two coercion variables $u : (\alpha \to \alpha) \multimap \alpha$ and $v : \alpha \multimap \alpha \to \alpha$ to model the recursive equality $o \cong o \to o$ of untyped λ-calculus. It is then straightforward to produce an $\mathsf{F_c}$ term a^* typable with α and such that $|a^*| = a$ for any untyped λ-term a:

$$x^* := x, \quad (ab)^* := (v \triangleright a^*)b^*, \quad (\lambda x.a)^* = u \triangleright (\lambda x.a^*). \qquad \square$$

It turns out that the solution we proposed in [19] was faulty. We tried there to ensure bisimulation by requiring coercion variables to have the restricted type $\tau \multimap \alpha$. As it turns out, this breaks subject reduction, as shown below with a counterexample due to Julien Cretin.

Consider the system $\mathsf{F_c}$ with the restriction that all coercion variables in the regular contexts have types of the form $\tau \multimap \alpha$. Let us define the context $\Gamma = \{x : \forall\alpha.\alpha \to \alpha \to \alpha, f : \beta \to \beta\}$ and the $\mathsf{F_c}$ term $a = (\underline{\measuredangle} c_1.\underline{\measuredangle} c_2.x(c_1 \triangleright f)(c_2 \triangleright f)) \triangleleft (\underline{\lambda} z.z)$. It is possible to prove that $\Gamma \vdash a : \mathrm{id}^\ell_{\mathrm{id}_\beta} \to \mathrm{id}_\beta$, where $\mathrm{id}_\beta = \beta \to \beta$ and $\mathrm{id}^\ell_\gamma = \gamma \multimap \gamma$, as follows (setting $\Gamma' = \Gamma, c_1 : \mathrm{id}^\ell_\gamma, c_2 : \mathrm{id}^\ell_\gamma$):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{matrix}\pi_1\\ \vdots\\ \Gamma';\vdash x(c_1 \triangleright f) : \gamma \to \gamma\end{matrix} \qquad \begin{matrix}\pi_2\\ \vdots\\ \Gamma';\vdash c_2 \triangleright f : \gamma\end{matrix}}{\Gamma' : \mathrm{id}^\ell_\gamma;\vdash x(c_1 \triangleright f)(c_2 \triangleright f) : \gamma}\textsc{App}}{\Gamma;\vdash \underline{\measuredangle} c_1.\underline{\measuredangle} c_2.x(c_1 \triangleright f)(c_2 \triangleright f) : \mathrm{id}^\ell_\gamma \to \mathrm{id}^\ell_\gamma \to \gamma}\textsc{CAbs}^2}{\Gamma;\vdash \underline{\measuredangle} c_1.\underline{\measuredangle} c_2.x(c_1 \triangleright f)(c_2 \triangleright f) : \forall\gamma.\mathrm{id}^\ell_\gamma \to \mathrm{id}^\ell_\gamma \to \gamma}\textsc{Gen}}{\Gamma;\vdash \underline{\measuredangle} c_1.\underline{\measuredangle} c_2.x(c_1 \triangleright f)(c_2 \triangleright f) : \mathrm{id}^\ell_{\mathrm{id}_\beta} \to \mathrm{id}^\ell_{\mathrm{id}_\beta} \to \mathrm{id}_\beta}\textsc{Inst} \qquad \dfrac{\dfrac{}{\Gamma; z : \mathrm{id}_\beta \vdash z : \mathrm{id}_\beta}\textsc{LAx}}{\Gamma;\vdash \underline{\lambda} z.z : \mathrm{id}^\ell_{\mathrm{id}_\beta}}\textsc{LAbs}}{\Gamma;\vdash a : \mathrm{id}^\ell_{\mathrm{id}_\beta} \to \mathrm{id}_\beta}\textsc{CApp}$$

In the proof above π_1 and π_2 are easy to obtain. If the subject reduction would hold, we would get also $\Gamma \vdash \underline{\measuredangle} c_2.xf(c_2 \triangleright f)$. However this judgment is not derivable because, to be able to abstract over c_2, its return type of should be both $\beta \to \beta$ and a type variable, which is impossible.

Indeed the reason why subject reduction does not hold is due to the failure of Lemma 9, which in turn is due to failure of the standard type subsitution property. In short, the condition on the context is not stable by type substitution, as the variable in the domain may be substituted by a full type.

In [15] a solution is given to this problem, by requiring that coercion variables are typed with variable codomain (or domain) and that those same type variables must be generalized right after the coercion variable gets abstracted. In this way one recovers the usual properties, including subject reduction, and bisimulation. We refer to that work for the details, while we will carry on the proof of strong normalization of all versions of $\mathsf{ML^F}$ by proving bisimulation directly for $\mathsf{xML^F}$.

4. Strong Normalization of $\mathsf{xML^F}$ via Translation

A translation from $\mathsf{xML^F}$ terms and instantiations into the coercion calculus is given in Figure 9. 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 $a\phi$ becomes a° coerced by ϕ°. One

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

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

Terms

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

Figure 9: Translation of types, instantiations and terms into the coercion calculus. For every type variable α we suppose fixed a fresh term variable i_α.

can see how this translation shares the same base idea as the one given for $\mathsf{iML^F}/\mathsf{eML^F}$ in [10].

We can already state how the translation "preserves semantics". As this concept is represented by type erasure in $\mathsf{xML^F}$ and coercion erasure in $\mathsf{F_c}$, it is achieved by the following easy result.

Lemma 25. *The type erasure of an* $\mathsf{xML^F}$ *term* a *coincides with the coercion erasure of its translation, i.e.* $\lceil a \rceil = \lfloor a^\circ \rfloor$.

Proof. Immediate induction. ◻

The rest of this section leads to the first main result of this work, namely SN of $\mathsf{xML^F}$. The same result for $\mathsf{eML^F}$ and $\mathsf{iML^F}$ will be established in the next section. We first need to show that the translation is sound from the point of view of typing. We will thus show that it maps typed terms to typed terms and typed instantiations to typed coercions.

Lemma 26. *If* $\Gamma \vdash \phi : \sigma \leq \tau$ *then* $\Gamma^\bullet; \vdash_{\mathsf{c}} \phi^\circ : \sigma^\bullet \multimap \tau^\bullet$.

Lemma 27. *If* a *is an* $\mathsf{xML^F}$ *term with* $\Gamma \vdash a : \sigma$ *then* $\Gamma^\bullet; \vdash_{\mathsf{t}} a^\circ : \sigma^\bullet$.

Lemma 28. *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) $(A\,[\phi;!\alpha/!\alpha])^\circ = A^\circ\,[(\underline{\lambda}z.i_\alpha \triangleright (\phi^\circ \triangleright z))/i_\alpha]$.

The above lemmas are proved by a standard induction. The interested reader can find their proofs in Appendix A.

Theorem 29 (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 $\mathsf{F_c}$.*

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 28(i).
- $a\mathbf{1} \to_\iota a$. We have $(a\mathbf{1})^\circ = \underline{\lambda} z.z \triangleright a^\circ \to_{\mathsf{c}} z\,[a^\circ/z] = a^\circ$.
- $a(\phi;\psi) \to_\iota a\phi\psi$. We have $(a(\phi;\psi))^\circ = (\underline{\lambda} z.\psi^\circ \triangleright (\phi^\circ \triangleright z)) \triangleright a^\circ \to_{\mathsf{c}} \psi^\circ \triangleright (\phi^\circ \triangleright a^\circ)$ which is equal to $(a\phi\psi)^\circ$.
- $a\mathbin{⅋} \to_\iota \Lambda(\alpha \geq \bot)a$. Here we have $(a⅋)^\circ = (\underline{\lambda} x.\underline{\lambda} i_\alpha.x) \triangleright a^\circ \to_{\mathsf{c}} \underline{\lambda} i_\alpha.a = (\Lambda(\alpha \geq \bot)a)^\circ$.
- $(\Lambda(\alpha \geq \tau)a)\& \to_\iota a\,[\mathbf{1}/!\alpha]\,[\tau/\alpha]$. Here, we have:

$$
\begin{aligned}
((\Lambda(\alpha \geq \tau)a)\&)^\circ &= (\underline{\lambda} x.x \triangleleft \underline{\lambda} z.z) \triangleright \underline{\measuredangle} i_\alpha.a^\circ \\
&\to_{\mathsf{c}} (\underline{\measuredangle} i_\alpha.a^\circ) \triangleleft \underline{\lambda} z.z \\
&\to_{\mathsf{c}} a^\circ\,[\underline{\lambda} z.z/i_\alpha] = (a\,[\mathbf{1}/!\alpha]\,[\tau/\alpha])^\circ, \text{ by Lemma 28(ii).}
\end{aligned}
$$

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

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

- $(\Lambda(\alpha \geq \tau)a)(\forall(\geq \phi)) \to_\iota \Lambda(\alpha \geq \tau\phi)a\,[\phi;!\alpha/!\alpha]$. We have:

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

  by Lemma 28(iii). ◻

Corollary 30 (Termination)**.** $\mathsf{xML^F}$ *is strongly normalizing.*

5. Transferring Strong Normalization from $\mathsf{xML^F}$ to $\mathsf{ML^F}$

In the previous section we have already shown SN of $\mathsf{xML^F}$. However in order to prove that $\mathsf{eML^F}$ and $\mathsf{iML^F}$ are normalizing too we need to make sure that ι-redexes cannot block β ones: in other words, a bisimulation result that we will achieve exploiting Theorem 31 below. Note that the type erasure of an $\mathsf{eML^F}$ term a can be defined analogously to the one for $\mathsf{xML^F}$ terms provided in Definition 3. Given an $\mathsf{eML^F}$ term a we still denoted by $\lceil a \rceil$ its type erasure (no confusion arises, since the context will disambiguate). From [8, Lemma 7, Theorem 6 and §4.2] we know the following[5].

Theorem 31. *For every $\mathsf{iML^F}$ (resp. $\mathsf{eML^F}$) term a, there is an $\mathsf{xML^F}$ term a^* such that $\lceil a^* \rceil = a$ (resp. $\lceil a^* \rceil = \lceil a \rceil$).*

In this section we provide a proof of Theorem 33, completely carried out within the $\mathsf{xML^F}$ system (given the SN result for $\mathsf{xML^F}$). We first need this intermediate lemma.

Lemma 32. *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 $a = \lambda(x : \tau)c$. $\square$

With the results above at hand we are ready to obtain the following weak bisimulation result.

Theorem 33 (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 $\lceil c \rceil = b$.*

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

[5] Notice that [8] uses the notation $[\![.]\!]$ for what we refer to with $(.)^*$.

Proof. 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 29). 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 32 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 $\lceil a_3\,[a_2/x] \rceil = d\,[\lceil a_2 \rceil / x] = b$. ☐

We are now ready to complete the main result of the paper for the other versions of $\mathsf{ML^F}$.

Corollary 34. *Terms typed in* $\mathsf{iML^F}$ *and* $\mathsf{eML^F}$ *are strongly normalizing.*

Proof. Suppose an $\mathsf{iML^F}$ term a has an infinite reduction. By Theorem 31 we have an $\mathsf{xML^F}$ term a^* such that $\lceil a^* \rceil = a$. Then by the bisimulation result above each step from a can iteratively be lifted to at least a step from a^*, giving rise to an infinite chain in $\mathsf{xML^F}$ which is impossible by Corollary 17.

For $\mathsf{eML^F}$ the reasoning is identical, there is only a further type erasure from $\mathsf{eML^F}$ to $\mathsf{iML^F}$. ☐

6. A Short Trip through Candidates of Reducibility

In this section we will show what results and difficulties one encounters if trying to adapt the proof by Girard and Tait's method of *candidates of reducibility* [4, 20] (or more precisely here *saturated sets*) to $\mathsf{ML^F}$. The base idea is analogous to what done for $\mathsf{F}_{<:}$ in [11]: in a nutshell, interpret the instance bound by a subset of candidates. However, one stumbles into a difficulty and an unexpected glitch which are worth mentioning.

- The method shows the strong normalization of $\lceil a \rceil$ for every $\mathsf{xML^F}$ term a, but cannot say anything about the non-trivial type reduction $\to_\iota$. A separate proof of SN of $\to_\iota$ is needed, which together with the bisimulation result of Theorem 33 gives then SN for the whole of $\to_{\beta\iota}$. Probably a direct proof of SN of $\to_\iota$ is not overtly hard, but the simulation to system F via $\mathsf{F_c}$ wraps SN of the whole of $\to_{\beta\iota}$ together.

- As one proves SN of $\lceil a \rceil$ for $\mathsf{xML^F}$ terms a, the result applies to $\mathsf{eML^F}$ or $\mathsf{iML^F}$ via compilation. However using the same interpretation directly on terms in $\mathsf{eML^F}/\mathsf{iML^F}$ and their types *does not work* in general. The apparent mismatch is due to the fact that the compilation a^* to $\mathsf{xML^F}$ described in [8] actually changes the type derivation of a before starting to build the $\mathsf{xML^F}$ term. So in fact there are some $\mathsf{iML^F}$ typings that do not survive the compilation process and which seem to pose serious issues to the candidates of reducibility argument. While we must admit it is quite confusing, we think this glitch may show some insight in $\mathsf{eML^F}$ and $\mathsf{iML^F}$'s type systems.

6.1. A Quick Recapitulation of Saturated Sets

We here briefly sketch the definitions and properties of *saturated sets* of ordinary λ-terms (whose set we denote by Λ). More details can be found in [21, 22]. We denote a sequence of terms $P_1 \cdots P_k$ by $\vec{P}$ and consequently the iterated application $MP_1 \cdots P_k$ by $M\vec{P}$.

Definition 35.

- Let $\mathsf{SN} := \{M \in \Lambda \mid M \text{ is strongly normalizable } \}$.
- For $\mathcal{A}, \mathcal{B} \subseteq \Lambda$ let $\mathcal{A} \to \mathcal{B} := \{M \in \Lambda \mid (\forall N \in A)\ MN \in B\}$.
- A set $\mathcal{A} \subseteq \mathsf{SN}$ is said to be **saturated** if

  S1) for all $\vec{P} \in \mathsf{SN}$ and any variable x we have $x\vec{P} \in \mathcal{A}$;

  S2) for all $\vec{P}, Q \in \mathsf{SN}$, if $M\,[Q/x]\,\vec{P} \in \mathcal{A}$ then $(\lambda x.M)Q\vec{P} \in \mathcal{A}$.

The set of saturated sets is denoted by SAT.

The following results are standard.

Lemma 36.

(i) SN *is saturated,*

(ii) $A, B \in \mathsf{SAT}$ *implies* $A \to B \in \mathsf{SAT}$*,*

(iii) *Given a family* $\{A_i\}_{i \in I}$ *such that* $A_i \in \mathsf{SAT}$ *we have* $\bigcap_{i \in I} A_i \in \mathsf{SAT}$*.*

6.2. Saturated Interpretation for $\mathsf{xML^F}$

In the following we will consider how to interpret types as saturated sets. As already hinted, the type instance relation $\leq$ will be modeled by set inclusion $\subseteq$ in SAT.

Definition 37. An **interpretation** Σ is a function from type variables to saturated sets. Let $\Sigma[\alpha \mapsto \mathcal{A}]$ be defined as Σ on $\beta \neq \alpha$ and as $\mathcal{A}$ on α. We extend an interpretation Σ to all $\mathsf{xML}^{\mathsf{F}}$ types by the following recursion:

$$\Sigma(\sigma \to \tau) := \Sigma(\sigma) \to \Sigma(\tau),$$
$$\Sigma(\forall(\alpha \geq \sigma)\tau) := \bigcap_{\substack{\mathcal{A}\in\mathsf{SAT}\\ \mathcal{A}\supseteq\Sigma(\sigma)}} \Sigma[\alpha \mapsto \mathcal{A}](\tau), \qquad \Sigma(\bot) := \bigcap_{\mathcal{A}\in\mathsf{SAT}} \mathcal{A}.$$

Lemma 36 shows that indeed the above definition maps types to SAT.

The following lemma is also quite standard and shown by a trivial induction.

Lemma 38.

(i) If $\alpha \notin \mathrm{ftv}(\sigma)$ then $\Sigma[\alpha \mapsto \mathcal{A}](\sigma) = \Sigma(\sigma)$;

(ii) $\Sigma(\sigma\,[\tau/\alpha]) = \Sigma[\alpha \mapsto \Sigma(\tau)](\sigma)$.

Definition 39. A **substitution** S is a function from term variables to *ordinary* λ-terms, which is then extended to all λ-terms by setting

$$S(M) = M\,[S(x_1)/x_1]\cdots[S(x_n)/x_n] \text{ where } \{x_1,\dots,x_n\} = \mathrm{fv}(M).$$

Given a subsitution S and an evaluation Σ, we write

- $\Sigma, S \vDash M : \sigma$ for an $\mathsf{xML}^{\mathsf{F}}$ term M if $S(\lceil M \rceil) \in \Sigma(\sigma)$;
- $\Sigma, S \vDash \Gamma$ for an $\mathsf{xML}^{\mathsf{F}}$ context if
  - for all $x : \sigma \in \Gamma$ we have $\Sigma, S \vDash x : \sigma$, i.e. $S(x) \in \Sigma(\sigma)$;
  - for all $\alpha \geq \sigma \in \Gamma$ we have $\Sigma(\alpha) \supseteq \Sigma(\sigma)$.

We divide the adequacy of the interpretation with respect to the typing rules in two results: in one we settle instantiations, while the other is for terms.

Lemma 40. *If $\Gamma \vdash \phi : \sigma \leq \tau$ and $\Sigma, S \vDash \Gamma$ then $\Sigma(\sigma) \subseteq \Sigma(\tau)$.*

Proof. By induction on the derivation, splitting by cases on the last rule.

- IComp and IRef are trivial.
- IBot, $\Gamma \vdash \tau : \bot \leq \tau$. By definition $\Sigma(\bot)$ is the bottom element of the meet-semilattice SAT.
- IAbstr, $\Gamma \vdash !\alpha : \tau \leq \alpha$ where $\alpha \geq \tau \in \Gamma$. By definition of $\Sigma, S \vDash \Gamma$, we have $\Sigma(\tau) \subseteq \Sigma(\alpha)$.
- IUnder, $\Gamma \vdash \forall(\alpha \geq)\phi : \forall(\alpha \geq \sigma)\tau_1 \leq \forall(\alpha \geq \sigma)\tau_2$. By well-formedness of the context in $\Gamma, \alpha \geq \sigma \vdash \phi : \tau_1 \leq \tau_2$ we have $\alpha \notin \mathrm{ftv}(\Gamma)$. Hence from $\Sigma, S \vDash \Gamma$ and for any $\mathcal{A} \supseteq \Sigma(\sigma)$ we can deduce $\Sigma[\alpha \to \mathcal{A}], S \vDash \Gamma, \alpha \geq \sigma$ by Lemma 38(i). By inductive hypothesis we then have $\Sigma[\alpha \to \mathcal{A}](\tau_1) \subseteq \Sigma[\alpha \to \mathcal{A}](\tau_2)$ for all $\mathcal{A} \supseteq \Sigma(\sigma)$, so that

$$\Sigma(\forall(\alpha \geq \sigma)\tau_1) = \bigcap_{\mathcal{A}\supseteq\Sigma(\sigma)} \Sigma[\alpha \mapsto \mathcal{A}](\tau_1) \subseteq \bigcap_{\mathcal{A}\supseteq\Sigma(\sigma)} \Sigma[\alpha \mapsto \mathcal{A}](\tau_2) = \Sigma(\forall(\alpha \geq \sigma)\tau_2).$$

- IInside, $\Gamma \vdash \forall(\geq \phi) : \forall(\alpha \geq \tau_1)\sigma \leq \forall(\alpha \geq \tau_2)\sigma$. By inductive hypothesis $\Sigma(\tau_1) \subseteq \Sigma(\tau_2)$, so that

  $$\{\, \mathcal{A} \in \mathsf{SAT} \mid \mathcal{A} \supseteq \Sigma(\tau_1) \,\} \supseteq \{\, \mathcal{A} \in \mathsf{SAT} \mid \mathcal{A} \supseteq \Sigma(\tau_2) \,\}$$

  which entails

  $$\Sigma(\forall(\alpha \geq \tau_1)\sigma) = \bigcap_{\mathcal{A} \supseteq \Sigma(\sigma)} \Sigma[\alpha \mapsto \mathcal{A}](\tau_1) \subseteq \bigcap_{\mathcal{A} \supseteq \Sigma(\sigma)} \Sigma[\alpha \mapsto \mathcal{A}](\tau_2) = \Sigma(\forall(\alpha \geq \tau_2)\sigma).$$

- IIntro, $\Gamma \vdash \mathcal{V} : \tau \leq \forall(\alpha \geq \bot)\tau$ where $\alpha \notin \mathrm{ftv}(\tau)$. Lemma 38(i) entails

  $$\Sigma(\forall(\alpha \geq \bot)\tau) = \bigcap_{\mathcal{A} \in \mathsf{SAT}} \Sigma[\alpha \mapsto \mathcal{A}](\tau) = \bigcap_{\mathcal{A} \supseteq \Sigma(\sigma)} \Sigma(\tau) = \Sigma(\tau).$$

- IElim, $\Gamma \vdash \& : \forall(\alpha \geq \sigma)\tau \leq \tau\,[\sigma/\alpha]$. We have

  $$\Sigma(\forall(\alpha \geq \sigma)\tau) = \bigcap_{\mathcal{A} \supseteq \Sigma(\sigma)} \Sigma[\alpha \mapsto \mathcal{A}](\tau_1) \subseteq \Sigma[\alpha \mapsto \Sigma(\sigma)](\tau_2) = \Sigma(\forall(\alpha \geq \sigma)\tau_2).$$

  where the last equality comes from Lemma 38(ii). □

Lemma 41. *If $\Gamma \vdash a : \sigma$ and $\Sigma, S \vDash \Gamma$ then $\Sigma, S \vDash M : \sigma$.*

Proof. Again an induction on the derivation of $\Gamma \vdash a : \sigma$ settles the case. Var, Abs and App are as usual, but we include the cases for completeness.

- Var, $\Gamma \vdash x : \tau$, where $\Gamma(x) = \tau$. Directly from the definition $\Sigma, S \vDash \Gamma$.

- Abs, $\Gamma \vdash \lambda(x : \tau)a : \tau \to \sigma$. In order to show that $S(\lceil \lambda(x : \tau)a \rceil) \in \Sigma(\tau \to \sigma) = \Sigma(\tau) \to \Sigma(\sigma)$ we take any $b \in \Sigma(\tau)$. Without loss of generality we can set $S(x) = x$ and $x \notin \mathrm{fv}(b)$. Then clearly $\Sigma, S[x \mapsto b] \vDash \Gamma, x : \tau$, so that inductive hypothesis $S(\lceil a \rceil)\,[b/x] = S[x \mapsto b](\lceil M \rceil) \in \Sigma(\sigma)$. By definition of saturated set we obtain $S(\lceil \lambda(x : \tau)a \rceil)b = \lambda x.S(\lceil a \rceil))b \in \Sigma(\sigma)$ which concludes the case.

- App, $\Gamma \vdash ab : \tau$ with $\Gamma \vdash a : \sigma \to \tau$. By induction hypothesis we have $\lceil a \rceil \in \Sigma(\sigma) \to \Sigma(\tau)$ and $\lceil b \rceil \in \Sigma(\sigma)$, which by definition entails $\lceil ab \rceil = \lceil a \rceil \lceil b \rceil \in \Sigma(\tau)$.

- TApp, $\Gamma \vdash a\phi : \sigma$ with $\Gamma \vdash \phi : \tau \leq \sigma$. By Lemma 40 $\Sigma(\tau) \subseteq \Sigma(\sigma)$, and by inductive hypothesis we can obtain

  $$S(\lceil a\phi \rceil) = S(\lceil a \rceil) \in \Sigma(\tau) \subseteq \Sigma(\sigma).$$

  which concludes the proof. □

Corollary 42. *If $\Gamma \vdash a : \sigma$ then $\lceil a \rceil \in \mathsf{SN}$.*

Proof. It suffices to take $\Sigma(\alpha) = \mathsf{SN}$ for all α (which is correct by Lemma 36) and $S(x) = x$ for all x. Then necessarily $\Sigma, S \vDash \Gamma$ (as $x \in \mathsf{SN}$ and $\mathsf{SN} \supseteq \Sigma(\tau)$), so that by the above lemma we get $\lceil a \rceil = S(\lceil a \rceil) \in \Sigma(\sigma) \subseteq \mathsf{SN}$. $\square$

Corollary 43. *If a is a typed* $\mathsf{iML^F}$ *or* $\mathsf{eML^F}$ *term then a is strongly normalizing.*

Proof. Even if a is in $\mathsf{eML^F}$ its reductions are exactly those of $\lceil a \rceil$. In any case by Theorem 31 we have $\lceil a \rceil = \lceil a^* \rceil \in \mathsf{SN}$. $\square$

Notice however that a separate proof of SN of $\to_\iota$ is needed to obtain again the remaining main result about SN of $\mathsf{xML^F}$. This is one of the main reasons we preferred anyway the proof via translation, the other reason being the study of $\mathsf{F_c}$ which has its own interest in our view.

6.3. The Issue of the Interpretation in $\mathsf{eML^F}$ *and* $\mathsf{iML^F}$*.*

Here we will briefly sketch the problems one encounters when applying the interpretation depicted above directly in $\mathsf{eML^F}$ or $\mathsf{iML^F}$. For the sake of space we will not be able to completely present the systems. The interested reader is referred to the literature about $\mathsf{ML^F}$ [6, 12, 13, 7].

First, types in $\mathsf{eML^F}$ and $\mathsf{iML^F}$ are built also out of the *rigid quantification* $\forall(\alpha = \sigma)\tau$. The most sensible way to interpret it would be

$$\Sigma(\forall(\alpha = \sigma)\tau) = \Sigma[\alpha \mapsto \Sigma(\sigma)](\tau) = \Sigma(\sigma\,[\tau/\alpha]),$$

in accordance with the semantic meaning given to rigid quantification, which is needed for type inference only.

Contrary to $\mathsf{xML^F}$, the instance relation on types is tiered in three parts: an equivalence $\equiv$ (for relations such as commutation of quantifiers or such as $\forall(\alpha \geq \sigma)\alpha \equiv \sigma$), an *abstraction* relation $\sqsubseteq\!\!\!\!\cdot$ which pertains operations concerning the rigid quantifier (so that for example $\Gamma \vdash \sigma \mathrel{\dot\sqsubseteq} \alpha$ if $\alpha = \sigma \in \Gamma$) and finally the instance relation $\sqsubseteq$. One has

$$\equiv \;\subseteq\; \dot\sqsubseteq \;\subseteq\; \sqsubseteq, \qquad \sqsubseteq \cap \sqsupseteq \;=\; \equiv.$$

With respect to $\mathsf{xML^F}$ there is a subtle difference between $\sqsubseteq$ and $\leq$, paramount to type inference. In fact $\leq$ may be decomposed as

$$\sigma \leq \tau \iff \sigma \mathrel{\dot\sqsupseteq}\sqsubseteq\mathrel{\dot\sqsupseteq} \tau$$

using $\dot\sqsupseteq$, the inverse relation of $\dot\sqsubseteq$. The part $\sqsubseteq$ of $\leq$ is completely recoverable by the automatic type inferencer, and it is in fact the $\dot\sqsupseteq$ parts that need explicit annotations in $\mathsf{eML^F}$. Notice that $\dot\sqsubseteq$ from the point of view of full type instance will be contained both in $\leq$ and $\geq$, so it is in fact part of the equivalence relation associated with the preorder $\leq$. Semantically $\dot\sqsubseteq$ is thus a completely reversible operation, while it is irreversible *vis-à-vis* the inferencer.

Because of the above reasons it is to be expected that the interpretation should enjoy the following (supposing $\Sigma, S \vDash \Gamma$):

- if $\Gamma \vdash \sigma \equiv \tau$ then $\Sigma(\sigma) = \Sigma(\tau)$;

- if $\Gamma \vdash \sigma \mathrel{\sqsupseteq\!\!\sqsubseteq} \tau$ then $\Sigma(\sigma) = \Sigma(\tau)$;
- if $\Gamma \vdash \sigma \sqsubseteq \tau$ then $\Sigma(\sigma) \subseteq \Sigma(\tau)$.

In fact the point that fails is already the first. If σ is equivalent to a *monomorphic* type (i.e. quantifier free), then we have:

$$\alpha \geq \sigma \in \Gamma \implies \tau \equiv \tau\,[\sigma/\alpha]$$

by the Eq-Mono rule of [6] (or by the *similarity* relation in the graphic representation of ML^F types [13, Definition 5.3.12]). Now there is no way to pass from $\Sigma(\alpha) \supseteq \Sigma(\sigma)$ of the hypothesis $\Sigma, S \vDash \Gamma$ to $\Sigma(\tau) = \Sigma(\tau\,[\sigma/\alpha])$. In rough words, there is no way for the interpretation as we defined to distinguish between a truly polymorphic type and a monomorphic one.

While we did try to change the interpretation of types along several directions, we always found some of the rules failing. However presenting these trials is well outside the scope of this paper, also due to their failure.

Conclusions

Related works

The present work solves an open problem precisely stated in [8, section 2.3]. Regarding the ML^F framework, no previous work had breached this or the bisimulation result, though another proof of both has appeared in the meantime in [15].

As already explained, the base idea behind the translation to system F here presented was already at the base of [10], namely translating bounded quantification $\forall(\alpha \geq \sigma)\tau$ with $\forall\alpha.(\sigma \to \alpha) \to \tau$ (in system F types). However this work is the first that explores the dynamic content of such an idea, whereas [10] was centred on static issues, possibly also because the explicit language xML^F was not yet developed at the time.

The other central theme of this paper are coercions, which has extensively been studied in the literature. We will cite here [23, 24, 14] and the already mentioned [15], though this should not be taken as an exhaustive list.

F_η [23] extends system F's typing rules by closing them under η-expansion. This is in fact equivalent to allowing a particular type of type conversion, referred to as type containment. As such F_η can indeed be seen as a calculus with coercions. Main differences with F_c are the lack of abstraction over coercions and the presence of variant-contravariant arrow coercions, which we will discuss next when considering further works. $\mathsf{F}_{<:}$ [24] endows system F with subtyping relations and allows bounded quantification, but in reverse order with respect to ML^F (i.e. $\forall(\alpha \leq \sigma)\tau$). Again, subtyping can be modelled with explicit coercions: $\mathsf{F}_{<:}$ allows abstraction over them (by abstracting bounds just like in xML^F), has arrow coercions, but disallow type instantiation. In [14] $\mathsf{F}_{<:}$ is translated by using type intersections with explicit coercions (with arrow ones included).

Finally [15] presents a new system F_ι, subsuming the ones briefly depicted above and ours. The main difference between our system and theirs (and a difference that applies to other systems cited above as well) is that in F_ι coercions are in fact syntactic entities completely separated from terms, while in F_c it is up to the type system to distinguish them. However for the moment this is at the expense of arrow coercions (which however xML^F lacks).

Full F_ι suffers from the same drawback of F_c, that is coercion variables that may block regular reductions thus denying bisimulation. One of the solutions proposed in [15] can be viewed as an amended version of our previous, faulty, proposition [19]. In addition to constraining type variables as the codomain of coercions (though letting more generally to have coercions with parametric domain too), the further restriction is to always generalize such a type variable when the corresponding coercion variable is abstracted. As shown in [15], such restriction suffices to guarantee subject reduction and termination.

Further works

We were able to prove new results for ML^F (namely SN and bisimulation of xML^F with its type erasure) by passing through 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 ML^F [10], 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 [23] or when using subtyping [14]. As explained at the beginning of section 3, F_c was purposely tailored down to suit xML^F, stripping it of natural features.

First, the lack of bisimulation can be amended by employing the restriction described in [15]. In that direction a more general calculus could be the next aim in this line of research.

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 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 introducing separate coercion syntactic entities is under investigation.

Acknowledgements

We thank Didier Rémy and Julien Cretin for stimulating discussions and remarks and the anonymous referees for their useful comments.

References

[1] R. Milner, M. Tofte, D. Macqueen, The definition of Standard ML, MIT Press, Cambridge, MA, USA, ISBN 0262631814, 1997.

[2] R. Milner, A theory of type polymorphism in programming, Journal of Computer and System Sciences 17 (1978) 348–75.

[3] 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, 2005.

[4] J.-Y. Girard, Y. Lafont, P. Taylor, Proofs and Types, no. 7 in Cambridge tracts in theoretical computer science, Cambridge University Press, 1989.

[5] J. B. Wells, Typability and Type Checking in System F are Equivalent and Undecidable, Ann. Pure Appl. Logic 98 (1-3) (1999) 111–56.

[6] D. L. Botlan, D. Rémy, ML^F: raising ML to the power of system F, in: C. Runciman, O. Shivers (Eds.), Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003, ACM, ISBN 1-58113-756-7, 27–38, 2003.

[7] D. L. Botlan, D. Rémy, Recasting ML^F, Inf. Comput. 207 (6) (2009) 726–85.

[8] D. Rémy, B. Yakobowski, A Church-Style Intermediate Language for ML^F, URL http://gallium.inria.fr/~remy/mlf/xmlf@long2010.pdf, to appear, 2011.

[9] H. Barendregt, The lambda calculus, its syntax and semantics, no. 103 in Studies in Logic and the Foundations of Mathematics, North-Holland, second edn., 1984.

[10] D. Leijen, A type directed translation of ML^F to System F, in: R. Hinze, N. Ramsey (Eds.), Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007, ACM Press, ISBN 978-1-59593-815-2, 111–22, 2007.

[11] G. Ghelli, Termination of System F-bounded: A Complete Proof, Inf. Comput. 139 (1) (1997) 39–56.

[12] D. Le Botlan, ML^F : Une extension de ML avec polymorphisme de second ordre et instanciation implicite, Ph.D. thesis, École Polytechnique, Available at gallium.inria.fr/~remy/mlf/mlf.pdf, 2004.

[13] B. Yakobowski, Types et contraintes graphiques : polymorphisme de second ordre et inférence, Ph.D. thesis, Université Paris Diderot (Paris 7), Available at hal.inria.fr/tel-00357708/, 2008.

[14] K. Crary, Typed compilation of inclusive subtyping, in: Proc. of International Conference on Functional Programming (ICFP'00), ACM Press, 68–81, 2000.

[15] J. Cretin, D. Rémy, Extending System F with Abstraction over Erasable Coercions, Tech. Rep. RR-7587, INRIA, URL http://hal.inria.fr/inria-00582570/fr/, 2011.

[16] A. Barber, G. Plotkin, Dual intuitionistic linear logic, Technical Report LFCS-96-347, University of Edinburgh, 1997.

[17] J.-Y. Girard, Linear logic, Theoretical Computer Science 50 (1987) 1–102.

[18] P. Baillot, K. Terui, Light types for polynomial time computation in lambda calculus, Inf. Comput. 207 (1) (2009) 41–62.

[19] G. Manzonetto, P. Tranquilli, Harnessing ML^{F} with the Power of System F, in: P. Hlinený, A. Kucera (Eds.), MFCS, vol. 6281 of *Lecture Notes in Computer Science*, Springer, ISBN 978-3-642-15154-5, 525–36, 2010.

[20] W. W. Tait, Intentional interpretation of functionals of finite type I, Journal of Symbolic Logic 32 (1967) 198–212.

[21] J.-L. Krivine, Lambda-calculus, Types and Models, Ellis Horwood, New York, ISBN 0-13-062407-1, translated from the ed. Masson, 1990, French original, 1993.

[22] H. P. Barendregt, Lambda Calculi with Types, in: S. A. D. M. Gabbay, T. Maibaum (Eds.), Handbook of Logic in Computer Science, vol. 1, Oxford University Press, 117–309, 1992.

[23] J. C. Mitchell, Coercion and type inference, in: Proc. of 11^{th} symposium on Principles of programming languages (POPL'84), ACM, ISBN 0-89791-125-3, 175–85, 1984.

[24] L. Cardelli, S. Martini, J. C. Mitchell, A. Scedrov, An Extension of System F with Subtyping, Inf. Comput. 109 (1/2) (1994) 4–56.

A. Technical Proofs

This technical appendix is devoted to provide the proofs of Lemmas 26, 27 and 28. These proofs are not particularly difficult, but long and require the following preliminary lemma.

Lemma 44. *Let σ, τ be $\mathsf{xML}^{\mathsf{F}}$ types, then $(\sigma\,[\tau/\alpha])^\bullet = \sigma^\bullet\,[\tau^\bullet/\alpha]$.*

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 26. If $\Gamma \vdash \phi : \sigma \leq \tau$ then $\Gamma^\bullet;\vdash_{\mathsf{c}} \phi^\circ : \sigma^\bullet \multimap \tau^\bullet$.

Proof. 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 $L := x : \forall\alpha.(\sigma^\bullet \multimap \alpha) \to \tau_1^\bullet$.

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

- 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) 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 $\Gamma';\vdash_{\mathsf{c}} \underline{\lambda} z.i_\alpha \triangleright (\phi^\circ \triangleright z) : \tau_1^\bullet \multimap \alpha$.

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

- IIntro, $\Gamma \vdash \mathbin{⅋} : \tau \leq \forall(\alpha \geq \bot)\tau$ 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:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma^\bullet, i_\alpha : (\forall\beta.\beta) \multimap \alpha; x : \tau^\bullet \vdash_\ell x : \tau^\bullet}\textsc{LAx}}{\Gamma^\bullet; x : \tau^\bullet \vdash_\ell \underline{\Lambda} i_\alpha.x : ((\forall\beta.\beta) \multimap \alpha) \to \tau^\bullet}\textsc{CAbs}}{\Gamma^\bullet; x : \tau^\bullet \vdash_\ell \underline{\Lambda} i_\alpha.x : (\forall(\alpha \geq \bot)\tau)^\bullet}\textsc{Gen}}{\Gamma^\bullet;\vdash_{\mathsf{c}} \underline{\lambda} x.\underline{\Lambda} i_\alpha.x : \tau^\bullet \multimap (\forall(\alpha \geq \bot)\tau)^\bullet}\textsc{LAbs}$$

- IElim, $\Gamma \vdash \& : \forall(\alpha \geq \sigma)\tau \leq \tau\,[\sigma/\alpha]$. 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 44. Let $L := x : \forall\alpha.(\sigma^\bullet \multimap \alpha) \to \tau^\bullet$.

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

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

Lemma 27. If a is an $\mathsf{xML^F}$ term with $\Gamma \vdash a : \sigma$ then $\Gamma^\bullet;\vdash_{\mathtt{t}} a^\circ : \sigma^\bullet$.

Proof. 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 Lemma 26. 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 $\Gamma^\bullet; \vdash_{\mathtt{t}} \phi^\circ \triangleright a^\circ : \sigma^\bullet$. □

Lemma 28. 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) $(A\,[\phi; !\alpha/!\alpha])^\circ = A^\circ\,[(\underline{\lambda} z.i_\alpha \triangleright (\phi^\circ \triangleright z))/i_\alpha]$.

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

$$\begin{array}{rl}
(\forall(\geq \phi)\,[\mathbf{1}/!\alpha]\,[\tau/\alpha])^\circ & = (\forall(\geq \phi\,[\mathbf{1}/!\alpha]\,[\tau/\alpha]))^\circ \\
& = \underline{\lambda} x.\underline{\not\lambda} i_\beta . x \triangleleft (\underline{\lambda} z.i_\beta \triangleright ((\phi\,[\mathbf{1}/!\alpha]\,[\tau/\alpha])^\circ \triangleright z)) \\
\text{(inductive hypothesis)} & = \underline{\lambda} x.\underline{\not\lambda} i_\beta . x \triangleleft (\underline{\lambda} z.i_\beta \triangleright ((\phi^\circ\,[\underline{\lambda} z.z/i_\alpha]) \triangleright z)) \\
& = (\underline{\lambda} x.\underline{\not\lambda} i_\beta . x \triangleleft (\underline{\lambda} z.i_\beta \triangleright (\phi^\circ \triangleright z)))\,[\underline{\lambda} z.z/i_\alpha] \\
& = (\forall(\geq \phi))^\circ\,[\underline{\lambda} z.z/i_\alpha]\,.
\end{array}$$

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

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

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

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

- $A = \mathbf{1}$: 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

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

- $A = \forall(\geq \phi)$: we have

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

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

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

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

$$
\begin{array}{rl}
((\phi;\psi) & [\phi;!\alpha/!\alpha])^\circ \\
& = (\phi\,[\phi;!\alpha/!\alpha]\,;\psi\,[\phi;!\alpha/!\alpha])^\circ \\
& = \underline{\lambda} x.(\psi\,[\phi;!\alpha/!\alpha])^\circ \triangleright ((\phi\,[\phi;!\alpha/!\alpha])^\circ \triangleright x) \\
\text{(ind. hyp.)} & = \underline{\lambda} x.(\psi^\circ\,[\underline{\lambda} z.i_\alpha \triangleright (\phi^\circ \triangleright z)/i_\alpha]) \triangleright ((\phi^\circ\,[\underline{\lambda} z.i_\alpha \triangleright (\phi^\circ \triangleright z)/i_\alpha]) \triangleright x) \\
& = (\underline{\lambda} x.\psi^\circ \triangleright (\phi^\circ \triangleright x))\,[\underline{\lambda} z.i_\alpha \triangleright (\phi^\circ \triangleright z)/i_\alpha] \\
& = (\phi;\psi)^\circ\,[\underline{\lambda} z.i_\alpha \triangleright (\phi^\circ \triangleright z)/i_\alpha]\,.
\end{array}
$$

- $A = \mathbf{1}$: we have $(\mathbf{1}\,[\phi;!\alpha/!\alpha])^\circ = \mathbf{1}^\circ = \underline{\lambda} x.x = \mathbf{1}^\circ\,[\underline{\lambda} z.i_\alpha \triangleright (\phi^\circ \triangleright z)/i_\alpha]$. $\square$