We provide a strong normalization result for $\text{ML}^F$, a type system generalizing $\text{ML}$ with first-class polymorphism as in system $F$. The proof is achieved by translating $\text{ML}^F$ into a calculus of coercions, and showing that this calculus is just a decorated version of system $F$. Simulation results then entail strong normalization from the same property of system $F$.

**Introduction.** $\text{ML}^F$ is a type system for (extensions of) $\lambda$-calculus which enriches $\text{ML}$ with the first class polymorphism of system $F$, 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 $\text{ML}$, while still being conservative: $\text{ML}$ programs still typecheck without needing any annotation. An important feature are principal type schemata, lacking in system $F$, which are obtained by employing a downward bounded quantification $\forall (\alpha \geq \sigma) \tau$, the so-called flexible quantifier. Such a type intuitively denotes that $\tau$ may be instantiated to any $\tau\{\sigma'/\alpha\}$, provided that $\sigma'$ is an instantiation of $\sigma$.

As already pointed out, system $F$ is contained in $\text{ML}^F$. It is not yet known, but it is conjectured [3], that the inclusion is strict. This makes the question of strong normalization (SN, i.e. whether $\lambda$-terms typed in $\text{ML}^F$ always terminate) a non-trivial one, to which we answer positively in this work. The result is proved via a suitable simulation in system $F$, with additional structure dealing with the complex type instantiations possible in $\text{ML}^F$.

Our starting point is $\text{xML}^F$, the Church version of $\text{ML}^F$: here type inference (and the rigid quantifier $\forall (\alpha = \sigma) \tau$ we did not mention) is abandoned, with the aim of providing an internal language to which a compiler might map the surface language briefly presented above (denoted $\text{eML}^F$ from now on [4]). Compared to Church-style system $F$, the type reduction $\rightarrow_\tau$ of $\text{xML}^F$ is more complex, and may a priori cause unexpected glitches: it could cause non-termination, or block the reduction of a $\beta$-redex. To prove that none of this happens, we use as target language of our translation a decoration of system $F$, the coercion calculus, which in our opinion has its own interest. Indeed, $\text{xML}^F$ has syntactic entities (the instantiations $\phi$) which testify an instance relation between types, and it is not hard to regard them as coercions. The delicate point is that some of these instantiations (the “abstractions” $!\alpha$) behave in fact as variables, abstracted when introducing a bounded quantifier: in a way, $\forall (\alpha \geq \sigma) \tau$ expects a coercion from $\sigma$ to $\alpha$, whatever the choice for $\alpha$ may be.

A question that arises naturally is: what does it mean to be a coercion in this context? Our answer, which works for $\text{xML}^F$, is in the form of a type system [Figure 2]. In section 2 we will...
show the good properties enjoyed by coercion calculus. The generality of coercion calculus allows then to lift these results to xMLF via a translation \(^{2}\). The main idea of the translation is the same as the one shown for eMLF in [4], where however no dynamic property was provided. Here we finally produce a proof of SN for all versions of MLF. Moreover the bisimulation result for xMLF establishes once and for all that it can be used as an internal language for eMLF, as the additional type structure cannot block reductions of the intended program.

1 A short introduction to xMLF

The syntactic entities of xMLF are presented in Figure 1. Intuitively, \(\bot \equiv \forall \alpha . \alpha\) and \(\forall (\alpha \geq \sigma) \tau\) restricts the variable \(\alpha\) to range over instances of \(\sigma\) only. Instantiations generalize system F’s type application, by providing a way to instantiate from one type to another. A \texttt{let}\ construct is added mainly to accommodate the type reconstructor of eMLF; apart from type inference purposes, one could assume \((\texttt{let} x = a \text{in} b) = (\lambda (x : \sigma) b) a\), with \(\sigma\) the correct type of \(a\). Apart from the usual variable assignments \(x : \tau\), environments also contain type variable assignments \(\alpha \geq \tau\), which are abstracted by the type abstraction \(\Lambda (\alpha \geq \tau) a\).

Typing judgments are of the usual form \(\Gamma \vdash a : \sigma\) for terms, and \(\Gamma \vdash \phi : \sigma \leq \tau\) for instantiations. The latter means that \(\phi\) can take a term \(a\) of type \(\sigma\) to \(a\phi\) of type \(\tau\). For the sake of space, we will not present here the typing rules of instantiations and terms, for which we refer to [3], along a more detailed discussion about xMLF. Reduction rules are divided into \(\rightarrow_{\beta}\) (regular \(\beta\)-reductions) and \(\rightarrow_{t}\), reducing instantiations. The type \(\tau\phi\) is given by an inductive definition (which we will not give here) which computes the unique type such that \(\Gamma \vdash \phi : \tau \leq \tau\phi\), if \(\phi\) typechecks. We recall (from [3]) that both \(\rightarrow_{\beta}\) and \(\rightarrow_{t}\) enjoy subject reduction. Moreover, we denote by \([a]\) the straightforwardly defined type erasure that ignores all type and instantiation annotations and maps xMLF terms to ordinary \(\lambda\)-terms (with \texttt{let}).

2 The coercion calculus

The syntax, the type system and the reduction rules of the coercion calculus are introduced in Figure 2. The notion of coercion is captured by the type \(\tau \rightarrow \sigma\): the use of linear logic’s linear

\[\begin{align*}
\sigma, \tau & ::= \alpha \mid \sigma \rightarrow \tau \mid \bot \mid \forall (\alpha \geq \sigma) \tau & \quad \text{(types)} \\
\phi, \psi & ::= \tau \mid ! \alpha \mid \forall (\alpha \geq \phi) \mid \forall (\alpha \geq \psi) \mid \& \mid \forall \phi; \psi \mid 1 & \quad \text{(instantiations)} \\
a, b, c & ::= x \mid \lambda (x : \tau). a \mid ab \mid \Lambda (\alpha \geq \tau). a \mid a\phi \mid \text{let} x = a \text{in} b & \quad \text{(terms)} \\
\Gamma, \Delta & ::= \emptyset \mid \Gamma, \alpha \geq \tau \mid \Gamma, x : \tau & \quad \text{(environments)}
\end{align*}\]

\begin{figure}[h]
\begin{center}
\begin{tabular}{ll}
\hline
\multicolumn{2}{|c|}{Syntactic definitions} \\
\hline
\sigma, \tau & ::= \alpha \mid \sigma \rightarrow \tau \mid \bot \mid \forall (\alpha \geq \sigma) \tau \\
\phi, \psi & ::= \tau \mid ! \alpha \mid \forall (\alpha \geq \phi) \mid \forall (\alpha \geq \psi) \mid \& \mid \forall \phi; \psi \mid 1 \\
a, b, c & ::= x \mid \lambda (x : \tau). a \mid ab \mid \Lambda (\alpha \geq \tau). a \mid a\phi \mid \text{let} x = a \text{in} b \\
\Gamma, \Delta & ::= \emptyset \mid \Gamma, \alpha \geq \tau \mid \Gamma, x : \tau \\
\hline
\multicolumn{2}{|c|}{Reduction rules} \\
\hline
(\lambda (x : \tau) a) b & \rightarrow_{\beta} a \{x/b\} & a \phi & \rightarrow_{t} (a\phi) \psi \\
\text{let} x = b \text{in} a & \rightarrow_{\beta} a \{x/b\} & (\Lambda (\alpha \geq \tau) a) \& \rightarrow_{t} a \{1/!\alpha\} \{\tau/\alpha\} & a(\phi; \psi) & \rightarrow_{t} (a\phi) \psi \\
(\Lambda (\alpha \geq \tau) a) (\forall (\alpha \geq \phi) \rightarrow, \Lambda (\alpha \geq \tau) (a\phi)) & \rightarrow_{t} (\Lambda (\alpha \geq \tau) a) (\forall (\geq \phi)) & a(\phi) \rightarrow_{t} (a\phi) \psi \\
\hline
\end{tabular}
\end{center}
\caption{Syntactic definitions and reduction rules of xMLF.}
\end{figure}
implication for the type of coercions is not casual. Indeed the typing system is a fragment of DILL, the dual intuitionistic linear logic [1]. This captures an aspect of coercions: they consume their argument without erasing it (as they must preserve it) nor duplicate it (as there is no true computation, just a type recasting). Environments are of shape $\Gamma$. Manzonetto & P. Tranquilli

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

### Syntactic definitions

| $\sigma, \tau := \alpha \mid \sigma \rightarrow \tau \mid \kappa \rightarrow \tau \mid \forall \alpha. \tau$ (types) | $\Gamma, \Delta := \emptyset \mid \Gamma, x : \tau \mid \Gamma, x : \sigma \rightarrow \alpha$ (environments) |
| $\kappa := \sigma \rightarrow \tau$ (coercion types) | $\Gamma : x : \tau$ (coercion environments) |
| $\zeta := \kappa \mid \kappa$ (type expressions) | $\Gamma ; \vdash a : \zeta$ (typing judgements) |
| $a, b ::= x \mid \lambda x.a \mid \lambda x.a \upharpoonright ab \mid a < b \mid a > b$ (terms) | $\Gamma ; \vdash \tau : a : \sigma$ (partial judgements) |
| $u, v ::= \lambda x.a \mid \lambda xu$ (c-values) | $\Gamma ; (z : \tau)^2$ stands for $\Gamma$; or $\Gamma ; z : \tau$. |

### Typing rules

| $\Gamma(y) = \zeta$ | $\Gamma ; \vdash a : \sigma \rightarrow \tau \mid \Gamma ; \vdash b : \sigma$ App |
| $\Gamma ; \vdash ab : \tau$ | $\Gamma ; (z : \tau)^2 \vdash a : \forall \alpha. \sigma$ INST |
| $\Gamma ; \vdash a : \tau \mid \Gamma ; x : \tau_1 ; \vdash b : \sigma$ LET |
| $\Gamma ; \vdash x : \tau \vdash a : \sigma$ ABS |
| $\Gamma ; \vdash \lambda x.a : \tau \rightarrow \sigma$ LAPP |
| $\Gamma ; \vdash \lambda x.a : \tau \rightarrow \sigma$ LAPP |
| $\Gamma ; \vdash a : \sigma_1 \rightarrow \sigma_2$ GEN |
| $\Gamma ; \vdash (\tau) \vdash a : \sigma$ GEN |
| $\Gamma ; \vdash \kappa : (\tau) \vdash a : \sigma$ GEN |
| $\Gamma ; (z : \tau)^2 \vdash \lambda x.a : \kappa \rightarrow \sigma$ CAPP |
| $\Gamma ; (z : \tau)^2 \vdash a < b : \sigma$ CAPP |

### Reduction rules

| $(\lambda x.a) \beta \rightarrow \beta a \{b/x\}$, | $\text{let } x = b \text{ in } a \beta \rightarrow \beta a \{b/x\}$, |
| $(\lambda x.a) \rightarrow_e a \{b/x\}$, | $(\lambda x.a) \rightarrow_e a \{b/x\}$, |
| $(\lambda x.a) \beta \rightarrow \beta a \{b/x\}$, | $(\lambda x.a) \rightarrow_e a \{b/x\}$, |

---

3Notice the restriction to $\sigma \rightarrow \alpha$ for coercion variables. Theorem 4 relies on this restriction $(d = \lambda x(e \rightarrow \delta)) \delta : (\sigma \rightarrow (\sigma \rightarrow \sigma)) \rightarrow \sigma$, with $\delta = \lambda y.y : \sigma$, $|d| = \delta \delta$ is a counterexample), but the preceding results do not.
of reductions can be proved by standard Tait-Martin Löl’s technique of parallel reductions. Summarizing, the following theorem holds.

**Theorem 2** (Confluence and termination). All of \( \to_\beta, \to_\varsigma, \to_\nu \) and \( \to_\beta^c \) are confluent. Moreover the coercion calculus is SN.

The use of coercions is annotated at the level of terms: \( \underline{\lambda} \) is used to distinguish between regular and coercion reduction, \( \triangleleft \) and \( \triangleright \) locate coercions without the need to carry typing information (the triangle’s side points out where the coercion is). Thus, the actual semantics of the term can be recovered via its coercion erasure:

\[
[x] := x, \quad [\lambda x.a] := \lambda x.[a], \quad [ab] := [a][b], \quad [\lambda x.a] := [a], \\
[\text{let} x = a \text{ in } b] = \text{let} x = [a] \text{ in } [b], \quad [a \triangleleft b] := [a], \quad [a \triangleright b] := [b].
\]

**Proposition 3** (Preservation of semantics). Take a typable coercion term \( a \). If \( a \to_\beta b \) (resp. \( a \to_\varsigma b \)) then \( [a] \to [b] \) (resp. \( [a] = [b] \)). Moreover we have the confluence diagram shown right.

The following result shows the connection between the reductions of a term and of its semantics.

**Theorem 4** (Bisimulation of \([.\] \)). If \( \Gamma ; \vdash A : \sigma \), then \( [a] \to_\beta [b] \) iff \( a \overset{\sigma}{\to_\varsigma} c \to_\beta c \) with \( [c] = b \).

### 3 The translation

A translation from xML\(^F \) terms and instantiations into the coercion calculus is given in Figure 3. The idea is that instantiations can be seen as coercions; thus a term starting with a type abstraction becomes a term waiting for a coercion, and a term \( a \phi \) becomes \( a^\circ \) coerced by \( \phi^\circ \). The rest of this section is devoted to showing how this translation and the properties of the coercion calculus lead to the main result of this work, SN of both xML\(^F \) and eML\(^F \). First one needs to show that the translation maps to typed terms. As expected, type instantiations are mapped to coercions.

**Proposition 5** (Soundness). For \( \Gamma ; \vdash a : \sigma \) an xML\(^F \) term (resp. \( \Gamma ; \vdash \phi : \sigma \leq \tau \) an xML\(^F \) instantiation) we have \( \Gamma^* ; \vdash a^\circ : \sigma^\circ \) (resp. \( \Gamma^* ; \vdash \phi^\circ : \sigma^\circ \to \tau^\circ \)). Moreover \( [a] = [a^\circ] \).

The following result shows that the translation is “faithful”, in the sense that \( \beta \) and \( \varsigma \) steps are mapped to \( \beta \) and \( \varsigma \) steps respectively: coercions do the job of instantiations, and just that.
**Proposition 6** (Coercion calculus simulates $xML^F$). If $a \rightarrow_\beta b$ (resp. $a \rightarrow_\iota b$) in $xML^F$, then $a^\circ \rightarrow_\beta b^\circ$ (resp. $a^\circ \vdash_\iota b^\circ$) in coercion calculus.

The above already shows SN of $xML^F$, however in order to show that $eML^F$ is also normalizing we need to make sure that $\iota$-redexes cannot block $\beta$ ones: in other words, a bisimulation result. The following is the lemma that does the trick, as it lets us lift to $xML^F$ the reduction in coercion calculus that bisimulates $\beta$-steps (Theorem 4).

**Lemma 7** (Lifting). For an $xML^F$ term $a$, if $a^\circ \rightsquigarrow_{cv} \rightarrow_\beta b$ then $b \rightarrow_\iota c$ with $a \rightarrow_\iota c$.

**Theorem 8** (Bisimulation of $\llbracket . \rrbracket$ for $xML^F$). For a typed $xML^F$ term $a$, we have that $\llbracket a \rrbracket \rightarrow_\beta b$ iff $a \rightarrow_\iota c$ with $\llbracket c \rrbracket = b$.

As a corollary of the two results stated above, we get the main result of this work, proving conclusively that all versions of $ML^F$ enjoy SN.

**Theorem 9** (SN of $ML^F$). Both $eML^F$ and $xML^F$ are strongly normalizing.

**Further work.** We were able to prove new results for $ML^F$ (namely SN and bisimulation of $xML^F$ with its type erasure) by employing a more general calculus of coercions. It becomes natural then to ask whether its typing system may be a framework to study coercions in general, like those arising in $F_\eta$ or when using subtyping. The typing rules of Figure 2 were purposely tailored down to $xML^F$ (for example disallowing in coercions polymorphism or coercion abstraction, i.e. coercion types $\forall \alpha. \kappa$ and $\kappa_1 \rightarrow \kappa_2$), stripping it of features which would not break the main results (though they would complicate their proofs).

Apart from such easy extensions we just mentioned, one would need a way to build coercions of arrow types, which are unneeded for $xML^F$. Namely, given coercions $c_1 : \sigma_2 \rightarrow \sigma_1$ and $c_2 : \tau_1 \rightarrow \tau_2$, there should be a coercion $c_1 \Rightarrow c_2 : (\sigma_1 \rightarrow \tau_1) \rightarrow (\sigma_2 \rightarrow \tau_2)$, allowing a reduction $(c_1 \Rightarrow c_2) \triangleright \lambda x.a \rightarrow_\iota \lambda x.c_2 \triangleright a \{c_1 \triangleright x / x\}$. This could be achieved by introducing it as a primitive, by translation or by special typing rules. Indeed if some sort of $\eta$-expansion would be available while building a coercion, one could write $c_1 \Rightarrow c_2 := \lambda f. \lambda x. (c_2 \triangleright (f(c_1 \triangleright x)))$. However how to do this without loosing bisimulation is under investigation.

**Acknowledgements.** We thank Didier Rémy for stimulating discussions and remarks.

**References**


