A Calculus of Coercions Proving the Strong Normalization of ML^F

Giulio Manzonetto*
LIPN, CNRS UMR 7030
Université Paris Nord, France
Giulio.Manzonetto@lipn.univ-paris13.fr

Paolo Tranquilli†
LIP, CNRS UMR 5668, INRIA
ENS de Lyon, Université Claude Bernard Lyon 1, France
Paolo.Tranquilli@ens-lyon.fr

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

Introduction. ML^F [3] is a type system for (extensions of) λ-calculus which enriches ML with the first class polymorphism of system F, providing a partial type annotation mechanism with an automatic type reconstructor. In this extension we can write programs that cannot be written in ML, while still being conservative: 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. This type says that τ may be instantiated to any $\tau\{\sigma'/\alpha\}$, *provided that* σ' *is an instantiation of* σ.

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

Our starting point is xML^F [5], the Church version of ML^F: here type inference (and the *rigid* quantifier $\forall(\alpha = \sigma)\tau$ we did not mention) is omitted, with the aim of providing an internal language to which a compiler might map the surface language briefly presented above (denoted eML^F from now on[1]). Compared to Church-style system F, the type reduction $\to_\iota$ of xML^F is more complex, and may *a priori* cause unexpected glitches: it could cause non-termination, or block the reduction of a β-redex. To prove that none of this happens, we use as target language of our translation a decoration of system F, the *coercion calculus*, which in our opinion has its own interest. Indeed, xML^F has syntactic entities (the *instantiations* ϕ) 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 fact, for all the choices of α, $\forall(\alpha \geq \sigma)\tau$ expects a coercion from σ to α.

A question that arises naturally is: what does it mean to be a coercion in this context? Our answer, which works for 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

*Supported by Digiteo project COLLODI (2009-28HD).
†Supported by ANR project COMPLICE (ANR-08-BLANC-0211-01).
[1]There is also a completely annotation-free version, iML^F, clearly at the cost of loosing type inference.

Submitted to:
HOR 2010

Syntactic definitions

$$\begin{array}{lll} \sigma,\tau & ::= \alpha \mid \sigma \to \tau \mid \bot \mid \forall(\alpha \geq \sigma)\tau & \text{(types)} \\ \phi,\psi & ::= \tau \mid !\alpha \mid \forall(\geq \phi) \mid \forall(\alpha \geq)\phi \mid \& \mid \parr \mid \phi;\psi \mid 1 & \text{(instantiations)} \\ a,b,c & ::= x \mid \lambda(x:\tau)a \mid ab \mid \Lambda(\alpha \geq \tau)a \mid a\phi \mid \mathtt{let}\, x = a\, \mathtt{in}\, b & \text{(terms)} \\ \Gamma,\Delta & ::= \emptyset \mid \Gamma, \alpha \geq \tau \mid \Gamma, x:\tau & \text{(environments)} \end{array}$$

Reduction rules

$$(\lambda(x:\tau)a)b \to_\beta a\{x/b\} \qquad a\parr \to_\iota \Lambda(\alpha \geq \bot)a, \quad \alpha \notin \mathrm{ftv}(\tau) \qquad a1 \to_\iota a$$

$$\mathtt{let}\, x = b\, \mathtt{in}\, a \to_\beta a\{x/b\} \qquad (\Lambda(\alpha \geq \tau)a)\& \to_\iota a\{1/!\alpha\}\{\tau/\alpha\} \qquad a(\phi;\psi) \to_\iota (a\phi)\psi$$

$$(\Lambda(\alpha \geq \tau)a)(\forall(\alpha \geq)\phi) \to_\iota \Lambda(\alpha \geq \tau)(a\phi) \qquad (\Lambda(\alpha \geq \tau)a)(\forall(\geq \phi)) \to_\iota \Lambda(\alpha \geq \tau\phi)a\{\phi;!\alpha/!\alpha\}$$

Figure 1: Syntactic definitions and reduction rules of xML^F.

then to lift these results to xML^F via a translation (section 3). The main idea of the translation is the same as the one shown for eML^F in [4], where however no dynamic property was provided. Here we finally produce a proof of SN for all versions of ML^F. Moreover the bisimulation result for xML^F establishes once and for all that xML^F can be used as an internal language for eML^F, as the additional type structure cannot block reductions of programs in eML^F.

1 A short introduction to xML^F

The syntactic entities of xML^F are presented in Figure 1. Intuitively, $\bot \cong \forall\alpha.\alpha$ and $\forall(\alpha \geq \sigma)\tau$ restricts the variable α to range over instances of σ only. Instantiations[2] generalize system F's type application, by providing a way to instantiate from one type to another. A `let` construct is added mainly to accommodate the type reconstructor of eML^F; apart from type inference purposes, one could assume $(\mathtt{let}\, x = a\, \mathtt{in}\, b) = (\lambda(x:\sigma)b)a$, with σ 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 ϕ can take a term a of type σ to $a\phi$ of type τ. For the sake of space, we will not present here the typing rules of instantiations and terms, for which we refer to [5], along with a more detailed discussion about xML^F. Reduction rules are divided into $\to_\beta$ (regular β-reductions) and $\to_\iota$, 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 ϕ typechecks. We recall (from [5]) that both $\to_\beta$ and $\to_\iota$ enjoy subject reduction. Moreover, we denote by $\lceil a \rceil$ the type erasure that ignores all type and instantiation annotations and maps xML^F terms to ordinary λ-terms (with `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 $\tau \multimap \sigma$: the use of linear logic's linear implication for the type of coercions is not casual. Indeed the typing system is a fragment of

[2] We follow the original notation of [5]; in particular it must be underlined that $\parr$ and & have no relation whatsoever with linear logic's par and with connectives.

Syntactic definitions

$\sigma,\tau ::= \alpha \mid \sigma \to \tau \mid \kappa \to \tau \mid \forall\alpha.\tau$ (types)
$\kappa ::= \sigma \multimap \tau$ (coercion types)
$\zeta ::= \tau \mid \kappa$ (type expressions)
$a,b ::= x \mid \lambda x.a \mid \underline{\lambda} x.a \mid ab \mid a \triangleleft b \mid a \triangleright b$ (terms)
$u,v ::= \lambda x.a \mid \underline{\lambda} x.u \mid x \triangleright u$ (c-values)

$\Gamma,\Delta ::= \emptyset \mid \Gamma, x:\tau \mid \Gamma, x:\sigma \multimap \alpha$ (environments)
$L ::= \emptyset \mid x:\tau$ (linear environments)
$\Gamma;\vdash a:\sigma$ (term judgements)
$\Gamma;\vdash a:\kappa$ (coercion judgements)
$\Gamma;z:\tau \vdash a:\sigma$ (linear judgements)

Typing rules

$$\frac{\Gamma(y)=\zeta}{\Gamma;\vdash y:\zeta}\textsc{Ax} \qquad \frac{\Gamma;\vdash a:\sigma\to\tau \quad \Gamma;\vdash b:\sigma}{\Gamma;\vdash ab:\tau}\textsc{App} \qquad \frac{\Gamma;L\vdash a:\forall\alpha.\sigma}{\Gamma;L\vdash a:\sigma\{\tau'/\alpha\}}\textsc{Inst}$$

$$\frac{\Gamma;\vdash a:\tau \quad \Gamma,x:\tau;\vdash b:\sigma}{\Gamma;\vdash \mathtt{let}\, x=a\,\mathtt{in}\, b:\sigma}\textsc{Let} \qquad \frac{\Gamma,x:\tau;\vdash a:\sigma}{\Gamma;\vdash \lambda x.a:\tau\to\sigma}\textsc{Abs} \qquad \frac{\Gamma;L\vdash a:\sigma \quad \alpha\notin \mathrm{ftv}(\Gamma;L)}{\Gamma;L\vdash a:\forall\alpha.\sigma}\textsc{Gen}$$

$$\frac{}{\Gamma;z:\tau\vdash z:\tau}\textsc{LAx} \qquad \frac{\Gamma;z:\tau\vdash a:\sigma}{\Gamma;\vdash \underline{\lambda} w.a:\tau\multimap\sigma}\textsc{LAbs} \qquad \frac{\Gamma;\vdash a:\sigma_1\multimap\sigma_2 \quad \Gamma;L\vdash b:\sigma_1}{\Gamma;L\vdash a\triangleright b:\sigma_2}\textsc{LApp}$$

$$\frac{\Gamma,x:\kappa;L\vdash a:\sigma}{\Gamma;L\vdash \underline{\lambda} x.a:\kappa\to\sigma}\textsc{CAbs} \qquad \frac{\Gamma;L\vdash a:\kappa\to\sigma \quad \Gamma;\vdash b:\kappa}{\Gamma;L\vdash a\triangleleft b:\sigma}\textsc{CApp}$$

Reduction rules

$$(\lambda x.a)b \to_\beta a\{b/x\}, \qquad \mathtt{let}\, x=b\,\mathtt{in}\, a \to_\beta a\{b/x\},$$
$$(\underline{\lambda} x.a)\triangleleft b \to_{\mathsf{c}} a\{b/x\}, \quad (\underline{\lambda} x.a)\triangleright b \to_{\mathsf{c}} a\{b/x\}, \quad (\underline{\lambda} x.u)\triangleleft b \to_{\mathsf{cv}} u\{b/x\}, \quad (\underline{\lambda} x.a)\triangleright u \to_{\mathsf{cv}} a\{u/x\}.$$

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

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 $\Gamma;L$, where Γ is a map from variables to type expressions[3], and L is the *linear* part of the environment, containing (contrary to DILL) *at most* one assignment.

Reductions are divided into $\to_\beta$ (the actual computation) and $\to_{\mathsf{c}}$ (the coercion reduction), having a subreduction $\to_{\mathsf{cv}}$ which intuitively is just enough to unlock β-redexes, and is needed for Theorem 4. We start from the basic properties of the coercion calculus. As usual, the following result is achieved with weakening and substitution lemmas.

Theorem 1 (Subject reduction)**.** *$\Gamma;L\vdash a:\zeta$ and $a\to_{\beta\mathsf{c}} b$ entail $\Gamma;L\vdash b:\zeta$.*

The coercion calculus can be seen as a decoration of Curry-style system F. The latter can be recovered by just collapsing the constructs $\multimap$, $\underline{\lambda}$, $\triangleleft$ and $\triangleright$ to their regular counterparts, via the *decoration erasure* defined as follows.

$$|\alpha| := \alpha, \quad |\zeta\to\tau| := |\zeta|\to|\tau|, \quad |\sigma\multimap\tau| := |\sigma|\to|\tau|, \quad |\Gamma|(y) := |\Gamma(y)|, \quad |\Gamma;z:\tau| := |\Gamma|, z:|\tau|,$$
$$|x| := x, \quad |\underline{\lambda} x.a| = |\lambda x.a| := \lambda x.|a|, \quad |\mathtt{let}\, x=a\,\mathtt{in}\, b| = (\lambda x.|b|)|a|, \quad |a\triangleleft b| = |a\triangleright b| = |ab| := |a||b|.$$

It is possible to prove that $\Gamma;L\vdash a:\zeta$ implies that $|\Gamma;L|\vdash a:|\zeta|$ in system F. From this, and the SN of system F [2, Sec. 14.3] it follows that the coercion calculus is SN. Confluence of reductions can be proved by standard Tait-Martin Löf's technique of parallel reductions. Summarizing, the following theorem holds.

[3]Notice the restriction to $\sigma\multimap\alpha$ for coercion variables. Theorem 4 relies on this restriction ($d=\underline{\lambda} x.(x\triangleright\delta)\delta : (\sigma\multimap(\sigma\to\sigma))\to\sigma$, with $\delta=\lambda y.yy:\sigma$, $\lfloor d\rfloor=\delta\delta$ is a counterexample), but the preceding results do not.

Types and contexts

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

Instantiations

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

Terms

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

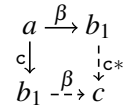
Figure 3: Translation of types, instantiations and terms into the coercion calculus. For every type variable α we suppose fixed a fresh term variable v_α.

Theorem 2 (Confluence and termination)**.** *All of $\to_\beta$, $\to_\mathsf{c}$, $\to_\mathsf{cv}$ and $\to_{\beta\mathsf{c}}$ 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 to the direction of the coercion). Thus, the actual semantics of the term can be recovered via its *coercion erasure*:

$$\lfloor x \rfloor := x, \qquad \lfloor \lambda x.a \rfloor := \lambda x.\lfloor a \rfloor, \qquad \lfloor ab \rfloor := \lfloor a \rfloor \lfloor b \rfloor, \qquad \lfloor \underline{\lambda}x.a \rfloor := \lfloor a \rfloor,$$
$$\lfloor \mathtt{let}\, x = a\, \mathtt{in}\, b \rfloor = \mathtt{let}\, x = \lfloor a \rfloor\, \mathtt{in}\, \lfloor b \rfloor, \qquad \lfloor a \triangleleft b \rfloor := \lfloor a \rfloor, \qquad \lfloor a \triangleright b \rfloor := \lfloor b \rfloor.$$

Proposition 3 (Preservation of semantics)**.** *Take a typable coercion term a. If $a \to_\beta b$ (resp. $a \to_\mathsf{c} b$) then $\lfloor a \rfloor \to \lfloor b \rfloor$ (resp. $\lfloor a \rfloor = \lfloor b \rfloor$). Moreover we have the confluence diagram shown on the right.*



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

Theorem 4 (Bisimulation of $\lfloor . \rfloor$)**.** *If $\Gamma;\vdash_A a : \sigma$, then $\lfloor a \rfloor \to_\beta b$ iff $a \xrightarrow{*}_\mathsf{cv} \to_\beta c$ with $\lfloor c \rfloor = b$.*

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° coerced by ϕ°. 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, that is SN of both xML^F and eML^F. First one needs to show that the translation maps to well-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 $\Gamma^\bullet;\vdash a^\circ : \sigma^\bullet$ (resp. $\Gamma^\bullet;\vdash \phi^\circ : \sigma^\bullet \multimap \tau^\bullet$). Moreover $\lceil a \rceil = \lfloor a^\circ \rfloor$.*

The following result shows that the translation is "faithful", in the sense that β and ι steps are mapped to β and c steps respectively: coercions do the job of instantiations, and just that.

Proposition 6 (Coercion calculus simulates xML^F)**.** *If $a \to_\beta b$ (resp. $a \to_\iota b$) in xML^F, then $a^\circ \to_\beta b^\circ$ (resp. $a^\circ \xrightarrow{+}_\mathsf{c} b^\circ$) 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 ι-redexes cannot block β ones: in other words, a bisimulation result. The following lemma lifts to xML^F the reduction in coercion calculus that bisimulates β-steps (Theorem 4).

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

Theorem 8 (Bisimulation of $\lceil . \rceil$ for xML^F)**.** *For a typed* xML^F *term* a*, we have that* $\lceil a \rceil \to_\beta b$ *iff* $a \xrightarrow{*}_\iota \to_\beta c$ *with* $\lceil c \rceil = b$*.*

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_η or when using subtyping. The typing rules of Figure 2 were tailored to xML^F, disallowing in coercions polymorphism or coercion abstraction, i.e. coercion types $\forall\alpha.\kappa$ and $\kappa_1 \to \kappa_2$. Removing such restrictions we could still derive the main result, even though the proofs would be more complex.

Apart from the extensions previously mentioned, one would need a way to build coercions of arrow types, which are unneeded for xML^F. Namely, given coercions $c_1 : \sigma_2 \multimap \sigma_1$ 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 by introducing it as a primitive, by translation or by special typing rules. Indeed if some sort of η-expansion would be available while building a coercion, one could write $c_1 \Rightarrow c_2 := \underline{\lambda} f.\lambda x.(c_2 \triangleright (f(c_1 \triangleright x)))$. However how to do this without loosing bisimulation is under investigation.

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

References

[1] Andrew Barber & Gordon Plotkin (1997): *Dual intuitionistic linear logic.* Technical Report LFCS-96-347, University of Edinburgh.

[2] Jean-Yves Girard, Yves Lafont & Paul Taylor (1989): *Proofs and Types.* Number 7 in Cambridge tracts in theoretical computer science. Cambridge University Press.

[3] Didier Le Botlan & Didier Rémy (2003): *MLF: Raising ML to the power of System F.* In: *Proc. of International Conference on Functional Programming (ICFP'03)*, pp. 27–38.

[4] Daan Leijen (2007): *A type directed translation of MLF to System F.* In: *Proc. of International Conference on Functional Programming (ICFP'07)*, ACM Press.

[5] Didier Rémy & Boris Yakobowski (2009): *A Church-style intermediate language for MLF.* Available at `http://www.yakobowski.org/xmlf.html`. Submitted.