A generalization of F_η with abstraction over retyping functions

Julien Cretin supervised by Didier Rémy
Gallium, INRIA

December 6, 2010

As stated on `http://mpri.master.univ-paris7.fr/stages.html`, this report is written in English because part of it will be incorporated into a research article written in English. As asked on the same website, here follows a two page synthesis of our work.

Synthesis

Context

Expressive languages often allow non trivial conversions between types, leading to complex, challenging, and sometimes ad hoc type systems. Such examples are the extension of System F with type equalities [3] to model Haskell GADTs and type families, or the extension of System F with explicit contracts. A useful technique to simplify the meta-theoretical studies of such systems is to make type injections fully explicit in terms via "coercions".

The essence of coercion functions is perhaps to be found in System F_η [1] which is the closure of System F by η-reduction. Indeed, this system can also be seen as the extension of System F with *retyping functions*. These retyping functions allow deep type specialization of terms, strengthening the domain of functions or weakening their codomains *a posteriori*. For example we have a retyping function from $(\tau[\alpha \leftarrow \tau']) \to (\forall\alpha.\sigma)$ to $(\forall\alpha.\tau) \to (\sigma[\alpha \leftarrow \sigma'])$.

We also find coercions in xML^F [5], which is the internal language of ML^F [4]. ML^F allows to have first-class polymorphism with inference and limited type annotations. xML^F is its Church-style variant.

Problem

Existing coercion systems propose different sets of features on coercions. For instance, F_η combines instantiation and contra-variance, xML^F has instantiation and abstraction, and $\mathsf{F}_{<:}$ [6] has contra-variance and abstraction. Instantiation is the possibility to go from $\forall\alpha.\tau$ to $\tau[\alpha \leftarrow \sigma]$. Contra-variance is the possibility to build a coercion like $(\tau' \to \sigma) \to (\tau \to \sigma')$, using smaller coercions like $\tau \to \tau'$ and $\sigma \to \sigma'$. Abstraction is the possibility to abstract on a coercion like $\lambda(c : \tau \to \tau').M$. We may wonder if there is a coercion system with all these features (instantiation, contra-variance, and abstraction). In other words, we want to generalize F_η with coercion abstraction, to see whether these features are compatible. This problematic was pointed out in previous work on xML^F as possible future work [2].

The key element of a language with coercions is that it has to preserve the coercion-erasure semantics. It means that coercions should have no run-time cost.

Solution

We describe a coercion system $\mathsf{F}_{\lambda\eta}$, which extends both F_η and xML^F. The system is mainly a colored version of Curry-style System F, where colors represent the coercions. We design the type system in order to ensure that coercions are retyping functions.

In this solution, coercions are usual lambda terms, as it is the case in [2]. We also tried another approach where coercions are proof terms of a type containment judgment, as it is the case in one of the language description in [1]. Showing an equivalence between these two descriptions, lambda term and proof term, is left for further work.

Most of the difficulty in the design of $\mathsf{F}_{\lambda\eta}$ is to preserve the coercion-erasure semantics, *i.e.* to allow coercions to be dropped before evaluation without changing the meaning of programs. Satisfying this property is particularly difficult in presence of instantiation and abstraction, and required to add some restrictions in the type system. But we believe some of these constraints can be softened. In particular the restriction on coercion abstractions can be easily modified as we see in future work.

Conclusion

This work shows that instantiation, contra-variance and abstraction are compatible within a coercion system. However, to preserve the coercion-erasure semantics, the restrictions we added disallow some interactions between contra-variance and abstraction. As a corollary, this work enforces the result of strong normalization of xML^F, which was not correctly demonstrated in [2].

Extensions

There are numerous paths to extend this work. First we can study systems which would be less restrictive on coercions abstractions, but still preserving the erasure semantics. An easy extension is to ask for parametricity either on the right of the arrow, as this is currently the case, or on the left. But, we would ideally want to completely remove this restriction, which would imply additional reduction rules, to avoid stuck terms because of coercion variables on arrow types.

Then, we might want to allow more colored function types. For example, the system could contain functions between coercions, or coercions between coercions, or even any function returning coercions.

Finally, we can look for a way to deduce a coercion from its type, and see if its normal form is unique. We can study how we could have inference in this system. And we can look for an equivalence proof between the current lambda description and a containment version of $\mathsf{F}_{\lambda\eta}$, in the same manner as [1].

Contents

1 Introduction

F_η and xML^F are two languages with coercions, which have overlapping but distinct specificities, and are not included into each other. Actually, F_η offers contra-variance which xML^F does not, and reciprocally, xML^F offers coercion abstraction which F_η does not. This naturally raises the question of the existence of a coercion language that unifies F_η and xML^F.

In this report, we design $\mathsf{F}_{\lambda\eta}$, a coercion language extending F_η with coercion abstraction. This language handles terms and coercions at the same level, so coercions are terms. We define a type system to ensure that coercions are just retyping functions [1]. A retyping function only changes the type of its argument. It cannot duplicate or erase it, and has to behave like the identity function. This allows us to say that coercions can be erased before running the program. Thus they have no run-time cost.

We recall the Lambda Calculus, System F, System F_η and xML^F definitions for the reader in Section 2. Our contributions are:

- to define the syntax, the type system and dynamic semantics of $\mathsf{F}_{\lambda\eta}$ in Section 3.1,

- to show that the type system is sound in Section 3.3,
- to show that $\mathsf{F}_{\lambda\eta}$ is strongly normalizing using the result of strong normalization in System F in Section 3.2,
- to show that both F_η and xML^F are included in $\mathsf{F}_{\lambda\eta}$ in Sections 4.1 and 4.2, and finally
- to show that $\mathsf{F}_{\lambda\eta}$ preserves the coercion erasure semantics in Section 4.3.

We defer until Section 5 the numerous ways of extending this work.

2 Existing Languages

To lighten the notation, we subscript the grammar rules of a language with a symbol denoting the language itself only in case of ambiguity. For example we have M_λ, M_F, M_η, M_x, and $M_{\lambda\eta}$ for the terms of the Lambda Calculus, System F, F_η, xML^F, and $\mathsf{F}_{\lambda\eta}$ respectively. We use the same term and type constructors for the languages. This way, we have obvious conversion from a language to the other, when their constructors match.

When inference rules are written in the flow of text, they do not contribute to the definition of the language. The languages are only defined in Figures.

2.1 Definition of the Lambda Calculus

We quickly recall the syntax and reduction relation of the Lambda Calculus with a Unit constructor in Figure 11 on page 21 in the Appendix. The reduction relation is strong, and we do not specify any reduction strategy. We use the Lambda Calculus as our target language when erasing coercions in $\mathsf{F}_{\lambda\eta}$.

2.2 Definition of F

We quickly recall the syntax, and the typing relation of Curry-style (no type annotations) System F in Figure 12 on page 22 in the Appendix. The reduction relation of System F is the same as the one of the Lambda Calculus. Note that the terms and contexts of the Lambda Calculus and System F are the same. We can see from the CONTEXT reduction rule that we use the strong reduction relation. Thus, the values are adapted to fit this description.

To simplify notations, we handle Γ_F as a set, and we can rename a bound variable in a term or a type at any time. We will do this kind of simplifications for the next languages too.

Like for the Lambda Calculus, this definition of System F has a Unit constructor. This will serve to show strong normalization of $\mathsf{F}_{\lambda\eta}$ in Section 3.2.

2.3 Definition of F_η

F_η allows to type more terms than System F. A term M is typeable and has type τ in F_η, if and only if there is a term N that η-reduces on M and which is typeable and has type τ in

System F. Said otherwise:

$$\frac{\Gamma \vdash_{\mathsf{F}} M : \tau \qquad M \rightsquigarrow_\eta N}{\Gamma \vdash_\eta N : \tau}$$

where η-reduction is:

$$\frac{M \rightsquigarrow_\eta M}{E[M] \rightsquigarrow_\eta E[M]}\ \textsc{Context} \qquad\qquad \lambda x.M\ x \rightsquigarrow_\eta M\ \textsc{Eta}$$

John C. Mitchell presented two versions of F_η and showed the equivalence between them [1]. The first version is exactly the intuition we have for F_η. It is defined as F with an additional typing rule that states:

$$\frac{\Gamma \vdash_\eta \lambda x.M\ x : \tau}{\Gamma \vdash_\eta M : \tau}$$

The second version uses a type containment judgment denoted by $\subseteq$ catching this notion of a term having several types. This judgment is used in a subtyping-like typing rule.

We only recall here the second version, which is the one we use in Section 4.1 to show the inclusion of F_η in $\mathsf{F}_{\lambda\eta}$. The syntax and the typing relation are defined in Figure 1.

When we have $\tau \subseteq \sigma$ then we can write a term M that has type $\tau \to \sigma$ in System F and that is an η-expansion of the identity function. This idea of a function being an η-expansion of the identity function $\lambda x.x$ is what Mitchell calls a *retyping function*, because the only thing that this function can do is change the type of its argument. We are going to extend this idea when designing $\mathsf{F}_{\lambda\eta}$.

2.4 Definition of $\mathsf{xML}^{\mathsf{F}}$

Curry-style system F is not used in practice, because it lacks inference. This comes from the first-class polymorphism which leads to undecidable type systems. The widespread ML solution is to have only second-class polymorphism. On the other hand, ML^{F} allows to have first-class polymorphism and still have inference by providing partial type annotations. We need to give the type of an argument if we use it in a polymorphic manner.

There are several variants of ML^{F}:

- $\mathsf{iML}^{\mathsf{F}}$, the Curry-style version, which does not need type annotations but does not have inference,
- $\mathsf{eML}^{\mathsf{F}}$, the version with type-inference, provided some annotations are prosent, and
- $\mathsf{xML}^{\mathsf{F}}$, the Church-style version, which has explicit type information.

Because inference is not yet one of our concern for $\mathsf{F}_{\lambda\eta}$, we will focus on $\mathsf{xML}^{\mathsf{F}}$. We give its definition in Figure 2, in almost the same manner as [2].

Figure 1: Definition of F_η

x, y	term variables
α, β	type variables
$M, N ::= x \mid \lambda x.M \mid M\ N$	terms
$\rho, \sigma, \tau ::= \alpha \mid \tau \to \sigma \mid \forall\alpha.\tau$	types
$A ::= \varnothing \mid A, x : \tau$	environments
$A \vdash_\eta M : \tau$	term judgments
$\tau \subseteq \sigma$	containment judgments

Definition of $\tau \subseteq \sigma$

(sub)
$$\frac{\overrightarrow{\beta} \notin \mathrm{ftv}(\overrightarrow{\forall\alpha.\sigma})}{\overrightarrow{\forall\alpha.\sigma} \subseteq \overrightarrow{\forall\beta.\sigma[\alpha \leftarrow \tau]}}$$

(dist)
$$\overrightarrow{\forall\alpha.(\tau \to \sigma)} \subseteq (\overrightarrow{\forall\alpha.\tau}) \to \overrightarrow{\forall\alpha.\sigma}$$

(arrow)
$$\frac{\sigma' \subseteq \sigma \qquad \tau \subseteq \tau'}{\sigma \to \tau \subseteq \sigma' \to \tau'}$$

(trans)
$$\frac{\rho \subseteq \sigma \qquad \sigma \subseteq \tau}{\rho \subseteq \tau}$$

(congruence)
$$\frac{\sigma \subseteq \tau}{\forall\alpha.\sigma \subseteq \forall\alpha.\tau}$$

Definition of $A \vdash_\eta M : \tau$

(var)
$$x : \sigma \vdash_\eta x : \sigma$$

(add hyp)
$$\frac{A \vdash_\eta M : \tau \qquad x \notin dom(A)}{A, x : \sigma \vdash_\eta M : \tau}$$

($\to I_\forall$)
$$\frac{A, x : \sigma \vdash_\eta M : \tau \qquad \overrightarrow{\alpha} \notin \mathrm{ftv}(A)}{A \vdash_\eta \lambda x.M : \overrightarrow{\forall\alpha.(\sigma \to \tau)}}$$

($\to E_\forall$)
$$\frac{A \vdash_\eta M : \overrightarrow{\forall\alpha.(\sigma \to \tau)} \qquad A \vdash_\eta N : \overrightarrow{\forall\alpha.\sigma}}{A \vdash_\eta M\ N : \overrightarrow{\forall\alpha.\tau}}$$

(cont)
$$\frac{A \vdash_\eta M : \sigma \qquad \sigma \subseteq \tau}{A \vdash_\eta M : \tau}$$

Figure 2: Definition of xML^F

$$
\begin{array}{rl}
x, y & \text{term variables} \\
\alpha, \beta & \text{type variables} \\
M, N ::= x \mid \lambda(x:\tau)M \mid M\,N \mid \Lambda(\alpha \geq \tau)M \mid M\,\phi & \\
\quad \mid \mathtt{let}\, x = M\, \mathtt{in}\, N & \text{terms} \\
\sigma, \tau ::= \alpha \mid \sigma \to \tau \mid \bot \mid \forall(\alpha \geq \sigma)\tau & \text{types} \\
\phi, \psi ::= \tau \mid \phi;\psi \mid \mathbf{1} \mid \& \mid \Re \mid !\alpha \mid \forall(\geq \phi) \mid \forall(\alpha \geq)\phi & \text{instantiations} \\
\Gamma ::= \varnothing \mid \Gamma, \alpha \geq \tau \mid \Gamma, x:\tau & \text{environments} \\
\Gamma \vdash_\mathsf{x} M : \tau & \text{term judgments} \\
\Gamma \vdash_\mathsf{x} \phi : \sigma \leq \tau & \text{instantiation judgments}
\end{array}
$$

Definition of $\Gamma \vdash_\mathsf{x} M : \tau$

$$\textsc{IBot}\quad \Gamma \vdash_\mathsf{x} \tau : \bot \leq \tau$$

$$\textsc{IAbs}\quad \Gamma, \alpha \geq \tau \vdash_\mathsf{x} !\alpha : \tau \leq \alpha$$

$$\textsc{IIntro}\quad \frac{\alpha \notin \mathrm{ftv}(\tau)}{\Gamma \vdash_\mathsf{x} \Re : \tau \leq \forall(\alpha \geq \bot)\tau}$$

$$\textsc{IComp}\quad \frac{\Gamma \vdash_\mathsf{x} \phi : \tau_1 \leq \tau_2 \qquad \Gamma \vdash_\mathsf{x} \psi : \tau_2 \leq \tau_3}{\Gamma \vdash_\mathsf{x} \phi;\psi : \tau_1 \leq \tau_3}$$

$$\textsc{IUnder}\quad \frac{\Gamma, \alpha \geq \tau \vdash_\mathsf{x} \phi : \tau_1 \leq \tau_2}{\Gamma \vdash_\mathsf{x} \forall(\alpha \geq)\phi : \forall(\alpha \geq \tau)\tau_1 \leq \forall(\alpha \geq \tau)\tau_2}$$

$$\textsc{IInside}\quad \frac{\Gamma \vdash_\mathsf{x} \phi : \tau_1 \leq \tau_2}{\Gamma \vdash_\mathsf{x} \forall(\geq \phi) : \forall(\alpha \geq \tau_1)\tau \leq \forall(\alpha \geq \tau_2)\tau}$$

$$\textsc{IElim}\quad \Gamma \vdash_\mathsf{x} \& : \forall(\alpha \geq \sigma)\tau \leq \tau[\alpha \leftarrow \sigma]$$

$$\textsc{IId}\quad \Gamma \vdash_\mathsf{x} \mathbf{1} : \tau \leq \tau$$

Definition of $\Gamma \vdash_\mathsf{x} \phi : \sigma \leq \tau$

$$\textsc{Var}\quad \Gamma, x:\tau \vdash_\mathsf{x} x : \tau$$

$$\textsc{Abs}\quad \frac{\Gamma, x:\tau \vdash_\mathsf{x} M : \sigma}{\Gamma \vdash_\mathsf{x} \lambda(x:\tau)M : \tau \to \sigma}$$

$$\textsc{App}\quad \frac{\Gamma \vdash_\mathsf{x} M : \tau \to \sigma \qquad \Gamma \vdash_\mathsf{x} N : \tau}{\Gamma \vdash_\mathsf{x} M\,N : \sigma}$$

$$\textsc{TAbs}\quad \frac{\Gamma, \alpha \geq \sigma \vdash_\mathsf{x} M : \tau \qquad \alpha \notin \mathrm{ftv}(\Gamma)}{\Gamma \vdash_\mathsf{x} \forall(\alpha \geq \sigma)M : \forall(\alpha \geq \sigma)\tau}$$

$$\textsc{TApp}\quad \frac{\Gamma \vdash_\mathsf{x} M : \tau \qquad \Gamma \vdash_\mathsf{x} \phi : \tau \leq \sigma}{\Gamma \vdash_\mathsf{x} M\,\phi : \sigma}$$

Figure 3: Drop Function

$$\lfloor \underline{x} \rfloor = x \qquad \lfloor \texttt{Unit} \rfloor = \texttt{Unit} \qquad \lfloor \underline{\lambda^{\square} x.M} \rfloor = \lfloor \underline{M} \rfloor \qquad \lfloor \underline{\lambda^{\square} x.M} \rfloor = \lambda x. \lfloor \underline{M} \rfloor \qquad \lfloor \underline{\lambda^{h} x.M} \rfloor = \lfloor \underline{M} \rfloor$$

$$\lfloor \underline{M} \mathbin{@^{\square}} \underline{N} \rfloor = \lfloor \underline{N} \rfloor \qquad \lfloor \underline{M} \mathbin{@^{\square}} \underline{N} \rfloor = \lfloor \underline{M} \rfloor \qquad \lfloor \underline{M} \mathbin{@^{\square}} \underline{N} \rfloor = \lfloor \underline{M} \rfloor \, \lfloor \underline{N} \rfloor \qquad \lfloor \underline{M} \mathbin{@^{h}} \underline{N} \rfloor = \lfloor \underline{M} \rfloor$$

3 Definition of $\mathsf{F}_{\lambda\eta}$

We make coercions explicit in the terms like in $\mathsf{xML}^{\mathsf{F}}$, but we also add them to the syntax of terms, in order to abstract over them. And we use the type system to ensure that these coercions only modify the type of their argument, and leave their semantics unchanged. This way, we can say they are *retyping functions* and that they do not compute. This is why we preserve the erasure semantics, and we can say that coercions have no run-time cost.

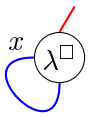
Our idea is to define a coercion, as an η-expansion of the identity function wrapped with erasable constructors inside and outside the η-expansions. This criterion is encoded into the type system in a way similar to [2].

We will mainly adapt Curry-style System F syntax, reduction rules and typing rules, by adding colors and flashes. So it is possible to read System F rules in background of our $\mathsf{F}_{\lambda\eta}$ rules. The new parts are colors, flashes and the return-type parametricity condition for coercion abstraction. Everything we add plays a role for the bisimulation result, which is what allows us to erase coercions and keep the same semantics.

3.1 Syntax

We define the syntax of $\mathsf{F}_{\lambda\eta}$ in Figure 4. Again, to simplify notations, we use Σ and Γ as sets, but we still use Δ as a list, and Z contains at most one element. This plays a key role in the type system, as this is how we render the η-expansion condition. We define by induction the drop fonction $\lfloor \underline{M} \rfloor$, from $\mathsf{F}_{\lambda\eta}$ to the Lambda Calculus in Figure 3. This function mainly just forgets about colors, flashes and coercions, and returns the Lambda Calculus skeleton of the term. It is only defined on normal terms, and not on coercions, because they are dropped.

Colors are part of the syntax, and can be seen as coloring the edges of the lambda term written as a tree. For example the identity coercion, which can only do top-level instantiation and generalization $\underline{\lambda^{\square} x.x}$ would be:



We use colors to pattern-match in the reduction rules and typing rules. They represent whether we are a normal term, or a coercion. Plain blue lines are used for normal terms, and dashed red lines for coercions. So $\underline{M}$ is a normal term, and $\underline{M}$ is a coercion. The fact that they are part of the syntax implies that there is a difference between $\underline{\lambda^{\square} x.M}$ and $\underline{\lambda^{\square} x.M}$.

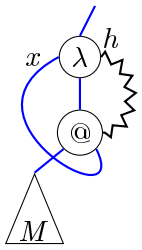
Figure 4: Syntax of $\mathsf{F}_{\lambda\eta}$

$$
\begin{array}{rll}
x, y & & \text{term variables} \\
h & & \text{flash variables} \\
\alpha, \beta & & \text{type variables} \\
\underline{\underline{\cdot}} & ::= \underline{\cdot} \mid \underline{\cdot}\!\cdot\!\underline{\cdot} & \text{colors} \\
\psi & ::= \Box \mid h & \text{flashes} \\
M, N, P & ::= x \mid \lambda^{\psi} \underline{\underline{x}}_1 . \underline{\underline{M}}_2 \mid \underline{\underline{M}}_1 @^{\psi} \underline{\underline{N}}_2 \mid \mathtt{Unit} & \text{terms} \\
\sigma, \tau & ::= \alpha \mid \underline{\underline{\sigma}}_1 \to \underline{\underline{\tau}}_2 \mid \forall \underline{\underline{\alpha}}_1 . \underline{\underline{\tau}}_2 \mid \mathtt{Unit} & \text{types} \\
p & ::= x \mid \underline{\underline{p}}_1 @^{\psi} \underline{\underline{v}}_2 & \text{prevalues} \\
v & ::= p \mid \lambda^{\psi} \underline{\underline{x}}_1 . \underline{\underline{v}}_2 \mid \mathtt{Unit} & \text{values} \\
E\,[\cdot] & ::= \cdot \mid \underline{\underline{\lambda^{\psi} \underline{\underline{x}}_2 . E\,[\cdot]}}_1 \mid \underline{\underline{E\,[\cdot] @^{\psi} \underline{\underline{M}}_2}}_1 \mid \underline{\underline{\underline{M}_2 @^{\psi} E\,[\cdot]}}_1 & \text{contexts} \\
\Sigma & ::= \varnothing \mid \Sigma, \underline{\underline{\alpha}} & \text{types environments} \\
\Gamma & ::= \varnothing \mid \Gamma, (\underline{\underline{x}}_1 : \underline{\underline{\tau}}_2) & \text{terms environments} \\
\mathrm{Z} & ::= \varnothing \mid \underline{\underline{x}}_1 : \underline{\underline{\tau}}_2 & \text{id environments} \\
\Delta & ::= \varnothing \mid \Delta, (h : \underline{\underline{x}}_1 : \underline{\underline{\tau}}_2) & \text{eta environments} \\
\Sigma; \Gamma; \mathrm{Z}; \Delta & \vdash \underline{\underline{M}}_1 : \underline{\underline{\tau}}_2 & \text{judgments}
\end{array}
$$

The first is a normal term abstraction returning a normal term, while the second is a coercion abstraction returning a normal term.

All the grammar rules but the one for contexts do not have a top-level color, we have to add it manually when we talk about them. For exemple M does not have a top-level color, whereas $\underline{M}$ does. Indeed, we see that $E\left[\underline{M}\right]$ does already have a top-level color in its definition. So we adopt the syntactic convention that $\underline{\underline{M}_2}_1 = \underline{M}_1 = \underline{M}_2$ which means that if we stack two colors for the same node (so the same edge), then these two colors are equal. This is used to do pattern-matching on the color, for instance when we write $\underline{E\left[\underline{M}_2\right]}_1$ instead of $E\left[\underline{M}\right]$ it means that the top-level color of $E\left[\underline{M}\right]$ has to be $\underline{\cdot}_1$. This does not happen very often, and it only happens in the presence of a context.

Flashes are used to keep track of η-expansions. They link a lambda node with a deeper application node that corresponds to the application node in the η-expansion. This explains the difference between $\underline{\lambda^{\square} x.M}$ and $\underline{\lambda^h x.M}$. The first is a normal abstraction, while the second is the abstraction part of an η-expansion. For the second to be well typed, we must have a $@^h$ appearing in $\underline{M}$. The simplest example involving flashes is the η-expansion of $\underline{M}$ which is $\underline{\lambda^h x.\underline{M}@^h x}$. Because flashes always go by pairs (a lambda node and an application node below), we say that these nodes are linked. Here is what looks like the tree of the last example (note the flash link between the lambda and application node):



Both term and flash variables are bound at the lambda in its only subterm. We can see it on the previous example. The flash variable is written above the lambda in the syntax, and the term variable is at its usual place before the point. The lambda does not always bind a flash variable, whereas it always binds a term variable. For instance, $\underline{\lambda^{\square} \underline{x}_1.\underline{M}_2}$ does not bind a flash variable, while $\underline{\lambda^h \underline{x}_1.\underline{M}_2}$ does. Flash variables can only be used at application nodes, but every application node does not necessarily have a flash variable. We write $\mathrm{ftv}(\cdot)$, $\mathrm{fv}(\cdot)$, and $\mathrm{ffv}(\cdot)$ for respectively the free type variables, the free term variables and the free flash variables functions. A typical example of a term using flashes is $\underline{\lambda^h x.E\left[\underline{M@^h N}\right]}$ where $x \notin \mathrm{fv}(\underline{M})$, $\underline{x}$ appears exactly once in $\underline{N}$, and $h \notin \mathrm{ffv}(\underline{M}, \underline{N}, \underline{E[\cdot]})$ which are enforced by typing. A picture of this term would be:

Figure 5: Substitution of $\mathsf{F}_{\lambda\eta}$

$$\underline{\underline{x}}_1[\underline{\underline{x}}_1 \leftarrow \underline{\underline{M}}_1] = \underline{\underline{M}}_1 \qquad \underline{\underline{y}}_1[\underline{\underline{x}}_2 \leftarrow \underline{\underline{M}}_2] = \underline{\underline{y}}_1$$

$$(\underline{\underline{\lambda^{\psi}\underline{\underline{y}}_2.\underline{\underline{M}}_3}}_1)[\underline{\underline{x}}_4 \leftarrow \underline{\underline{N}}_4] = \underline{\underline{\lambda^{\psi}\underline{\underline{y}}_2.\underline{\underline{M}}_3[\underline{\underline{x}}_4 \leftarrow \underline{\underline{N}}_4]}}_1 \text{ with } y \notin \mathrm{fv}(x, N) \text{ and } \psi \notin \mathrm{ffv}(N)$$

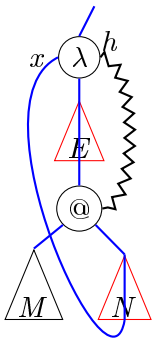
$$(\underline{\underline{\underline{\underline{M}}_2@^{\psi}\underline{\underline{N}}_3}}_1)[\underline{\underline{x}}_4 \leftarrow \underline{\underline{P}}_4] = \underline{\underline{\underline{\underline{M}}_2[\underline{\underline{x}}_4 \leftarrow \underline{\underline{P}}_4]@^{\psi}\underline{\underline{N}}_3[\underline{\underline{x}}_4 \leftarrow \underline{\underline{P}}_4]}}_1 \qquad \underline{\underline{\mathtt{Unit}}}_1[\underline{\underline{x}}_2 \leftarrow \underline{\underline{P}}_2] = \underline{\underline{\mathtt{Unit}}}_1$$

$$\underline{\underline{x}}_1[h \leftarrow \psi] = \underline{\underline{x}}_1 \qquad (\underline{\underline{\lambda^{\psi'}\underline{\underline{x}}_2.\underline{\underline{M}}_3}}_1)[h \leftarrow \psi] = \underline{\underline{\lambda^{\psi'}\underline{\underline{x}}_2.\underline{\underline{M}}_3[h \leftarrow \psi]}}_1 \text{ with } \psi' \notin \mathrm{ffv}(h, \psi)$$

$$(\underline{\underline{\underline{\underline{M}}_2@^{\psi'}\underline{\underline{N}}_3}}_1)[h \leftarrow \psi] = \begin{cases} \underline{\underline{M}}_2@^{\psi}\underline{\underline{N}}_3 & \text{if } \psi' = h \\ \underline{\underline{\underline{\underline{M}}_2[h \leftarrow \psi]@^{\psi'}\underline{\underline{N}}_3[h \leftarrow \psi]}}_1 & \text{otherwise} \end{cases} \qquad \underline{\underline{\mathtt{Unit}}}_1[h \leftarrow \psi] = \underline{\underline{\mathtt{Unit}}}_1$$

$$\underline{\underline{\alpha}}_1[\underline{\underline{\alpha}}_1 \leftarrow \underline{\underline{\sigma}}_1] = \underline{\underline{\sigma}}_1 \qquad \underline{\underline{\beta}}_1[\underline{\underline{\alpha}}_2 \leftarrow \underline{\underline{\sigma}}_2] = \underline{\underline{\beta}}_1 \qquad \underline{\underline{\underline{\underline{\tau}}_2 \to \underline{\underline{\tau'}}_3}}_1[\underline{\underline{\alpha}}_4 \leftarrow \underline{\underline{\sigma}}_4] = \underline{\underline{\underline{\underline{\tau}}_2[\underline{\underline{\alpha}}_4 \leftarrow \underline{\underline{\sigma}}_4] \to \underline{\underline{\tau'}}_3[\underline{\underline{\alpha}}_4 \leftarrow \underline{\underline{\sigma}}_4]}}_1$$

$$(\underline{\underline{\forall\underline{\underline{\beta}}_2.\underline{\underline{\tau}}_3}}_1)[\underline{\underline{\alpha}}_4 \leftarrow \underline{\underline{\sigma}}_4] = \underline{\underline{\forall\underline{\underline{\beta}}_2.\underline{\underline{\tau}}_3[\underline{\underline{\alpha}}_4 \leftarrow \underline{\underline{\sigma}}_4]}}_1 \text{ with } \beta \notin \mathrm{ftv}(\alpha, \sigma) \qquad \underline{\underline{\mathtt{Unit}}}_1[\underline{\underline{\alpha}}_2 \leftarrow \underline{\underline{\sigma}}_2] = \underline{\underline{\mathtt{Unit}}}_1$$



The meaning of the red triangles is that they are an erasable wrapping of the term going through them. More precisely $E[\cdot]$ is an erasable wrapping if $\lfloor E[\cdot] \rfloor = \lfloor \cdot \rfloor$.

We define substitutions for terms, flashes and types in Figure 5. The side conditions we have for lambdas are obviously always satisfied, because we can always rename the bound variables to avoid the conflict.

We define the reduction relation in Figure 6. We subscript our reductions with β or ι depending on whether the reduction is a computation step or not, respectively. A reduction is ι if its redex contains colors or flashes, and β otherwise. All the reduction rules but NIotaUp are the usual β-reduction modulo colors and flashes. NIotaUp is a β-reduction under a context catching the binder of the flash on the application node. The intuition about flashes during reduction comes from the fact that we do not want the reduction involving a flash to count as a computation step, because it was just an η-expansion in a retyping function.

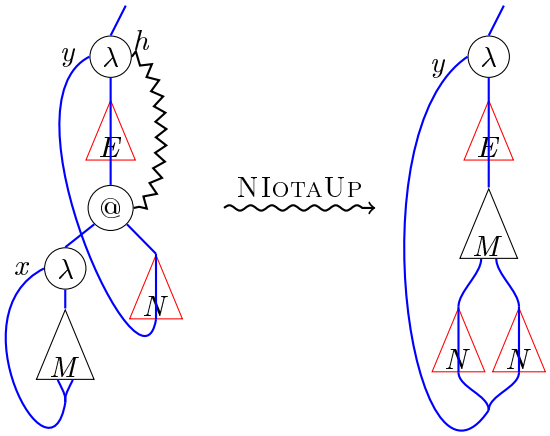
Figure 6: Reduction relation of $\mathsf{F}_{\lambda\eta}$

$$\textsc{ContextBeta}\ \frac{M \rightsquigarrow_\beta N}{E\,[M] \rightsquigarrow_\beta E\,[N]} \qquad \textsc{Beta}\ (\lambda^\square x.M)@^\square N \rightsquigarrow_\beta M[x \leftarrow N] \qquad \textsc{ContextIota}\ \frac{M \rightsquigarrow_\iota N}{E\,[M] \rightsquigarrow_\iota E\,[N]}$$

$$\textsc{CIota}\ (\lambda^\square x.M)@^\square N \rightsquigarrow_\iota M[x \leftarrow N] \qquad \textsc{LIota}\ (\lambda^\square x.M)@^\square N \rightsquigarrow_\iota M[x \leftarrow N]$$

$$\textsc{NIotaDown}\ (\lambda^h x.M)@^\psi N \rightsquigarrow_\iota M[h \leftarrow \psi][x \leftarrow N] \qquad \textsc{NIotaUp}\ \lambda^h y.E\left[(\lambda^\square x.M)@^h N\right] \rightsquigarrow_\iota \lambda^\square y.E\,[M[x \leftarrow N]]$$

However in the current redex it is the lambda node which is flashed, then, there will be another redex with the associated flashed application node, and this time it should be a computation step. So when a redex contains a flash, we remove its associated flash. And when both the lambda and application nodes of a redex are flashed, we link their associated flashes together. This way η-expansions count as ι-reduction, but still leave one β-reduction, because n nested η-expansions spread over $n+1$ redexes. So we want the n first reduction to be ι-reductions, and the last one to be a β-reduction. Because we did not specify any reduction strategy, this system of flashes allows us to always have a well defined drop function according to the bisimulation.



We see on the illustration of NIotaUp that the value of drop on both side is the same. This is what we want because this is a ιstep. Also note that in a redex, the colors match between the bound variable and the argument, and between the body and the whole term.

Here are how we understand the different sort of environments, that are used later in the typing judgment.

- Σ contains and "binds" the type variables, so the judgment is closed for type variables.

- Γ contains and "binds" some term variables, and it also remembers the type of each variable. This is the usual environment, because it's a set and we do not use it in any particular way in the typing rules.
- Z is either empty or contains and "binds" exactly one term variable with its type. When $\mathrm{Z} = \varnothing$, we don't have any particular constraint. But when $\mathrm{Z} = \underline{x} : \underline{\tau}$, it means the term we are typing erases to $\underline{x}$, so it is $E\left[\underline{x}\right]$ with $E\left[\cdot\right]$ an erasable wrapping. This is what ensures retyping functions to behave like the identity function (see the LApp typing rule).
- Δ contains and "binds" the remaining term variables, so the judgment is closed for term variables. Each element of the list is a triple containing a flash variable, a term variable and its type. The intuition of $(h_1 : \underline{x_1} : \underline{\tau_1}), \ldots, (h_n : \underline{x_n} : \underline{\tau_n})$ is that we went through n flashed lambdas successively. Each of these flashed lambda is the lambda of an η-expansion. They are stacked, and waiting to be unstacked by their corresponding application node.

We define the typing relation by induction in Figure 7. We see these environments in action in the XAbs and XApp rules, which are in four versions.

- Abs and App are for normal terms abstraction. Z and Δ are empty because $\underline{\lambda^{\square} \underline{x}.\underline{M}}$ and $\underline{\underline{M} @^{\square} \underline{N}}$ are involved in computation, i.e. β-reduction. These two constructors constitute the skeleton of a term, which remains after coercion-erasure. The following constructors are here to wrap the skeleton and change its type.
- CAbs and CApp are for coercions abstraction, and we can see the return-type parametricity condition that we need to show the bisimulation. This time Z and Δ may not be empty, because we are just wrapping the term, i.e. the constructors we add are going to be erased.
- LAbs and LApp ensure that coercions behave as the identity function. Z and Δ are empty when typing a coercion, because coercions are dropped, and we do not want to drop resources.
- NAbs and NApp allow to do η-expansions at no run-time cost. We see in NApp that $\underline{N}$ should be an erasable wrapping of $\underline{x}$, and that $\underline{x}$ does not appear in $\underline{M}$. This is how we ensure flashed terms to be η-expansions wrapped with coercions and other η-expansions.

3.2 Strong Normalization

We define $\widehat{\underline{\underline{M}}}$ from $\mathsf{F}_{\lambda\eta}$ to F by induction in Figure 8. This function forgets about the colors and the flashes, but coercions become normal terms of System F.

First we show that each step in $\mathsf{F}_{\lambda\eta}$ corresponds to a step in F. Then we show that the translation of a well typed term in $\mathsf{F}_{\lambda\eta}$ is well typed in F. Finally, we get back that $\mathsf{F}_{\lambda\eta}$ is strongly normalizing for well typed terms. We use this result in the bisimulation proof to show that $\rightsquigarrow_\iota$ is strongly normalizing.

Figure 7: Typing relation of $\mathsf{F}_{\lambda\eta}$

$$\textsc{TypeVariable}\quad \Sigma, \alpha \vdash \alpha$$

$$\textsc{TypeArrow}\quad \frac{\Sigma \vdash \tau_2 \qquad \Sigma \vdash \sigma_3}{\Sigma \vdash \tau_2 \to \sigma_{3_1}}$$

$$\textsc{TypeForall}\quad \frac{\Sigma, \alpha \vdash \tau_2}{\Sigma \vdash \forall \alpha.\tau_{2_1}}$$

$$\textsc{TypeUnit}\quad \frac{}{\Sigma \vdash \texttt{Unit}}$$

$$\textsc{EnvEmpty}\quad \Sigma \vdash \varnothing$$

$$\textsc{EnvVariable}\quad \frac{\Sigma \vdash \Gamma \qquad \Sigma \vdash \tau \qquad x \notin \mathrm{fv}(\Gamma)}{\Sigma \vdash \Gamma, (x : \tau)}$$

$$\textsc{Ax}\quad \frac{\Sigma \vdash \Gamma, (x : \tau)}{\Sigma; \Gamma, (x : \tau); \varnothing; \varnothing \vdash x : \tau}$$

$$\textsc{LAx}\quad \frac{\Sigma \vdash \Gamma, (x : \tau)}{\Sigma; \Gamma; x : \tau; \varnothing \vdash x : \tau}$$

$$\textsc{Unit}\quad \frac{\Sigma \vdash \Gamma}{\Sigma; \Gamma; \varnothing; \varnothing \vdash \texttt{Unit} : \texttt{Unit}}$$

$$\textsc{Abs}\quad \frac{\Sigma; \Gamma, (x : \tau); \varnothing; \varnothing \vdash M : \sigma}{\Sigma; \Gamma; \varnothing; \varnothing \vdash \lambda^{\square} x.M : \tau \to \sigma}$$

$$\textsc{App}\quad \frac{\Sigma; \Gamma; \varnothing; \varnothing \vdash M : \tau \to \sigma \qquad \Sigma; \Gamma; \varnothing; \varnothing \vdash N : \tau}{\Sigma; \Gamma; \varnothing; \varnothing \vdash M @^{\square} N : \sigma}$$

$$\textsc{CAbs}\quad \frac{\Sigma, \alpha; \Gamma, (x : \tau \to \alpha); \mathrm{Z}; \Delta \vdash M : \sigma}{\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash \lambda^{\square} x.M : \forall \alpha.(\tau \to \alpha) \to \sigma}$$

$$\textsc{CApp}\quad \frac{\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash M : \tau \to \sigma \qquad \Sigma; \Gamma; \varnothing; \varnothing \vdash N : \tau}{\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash M @^{\square} N : \sigma}$$

$$\textsc{LAbs}\quad \frac{\Sigma; \Gamma; x : \tau; \varnothing \vdash M : \sigma}{\Sigma; \Gamma; \varnothing; \varnothing \vdash \lambda^{\square} x.M : \tau \to \sigma}$$

$$\textsc{LApp}\quad \frac{\Sigma; \Gamma; \varnothing; \varnothing \vdash M : \tau \to \sigma \qquad \Sigma; \Gamma; \mathrm{Z}; \Delta \vdash N : \tau}{\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash M @^{\square} N : \sigma}$$

$$\textsc{NAbs}\quad \frac{\Sigma; \Gamma; \mathrm{Z}; \Delta, (h : x : \tau) \vdash M : \sigma}{\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash \lambda^{h} x.M : \tau \to \sigma}$$

$$\textsc{NApp}\quad \frac{\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash M : \tau' \to \sigma \qquad \Sigma; \Gamma; x : \tau; \varnothing \vdash N : \tau'}{\Sigma; \Gamma; \mathrm{Z}; \Delta, (h : x : \tau) \vdash M @^{h} N : \sigma}$$

$$\textsc{TAbs}\quad \frac{\Sigma, \alpha; \Gamma; \mathrm{Z}; \Delta \vdash M : \tau}{\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash M : \forall \alpha.\tau}$$

$$\textsc{TApp}\quad \frac{\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash M : \forall \alpha.\tau \qquad \Sigma \vdash \sigma}{\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash M : \tau[\alpha \leftarrow \sigma]}$$

Figure 8: Translation from $\mathsf{F}_{\lambda\eta}$ to F

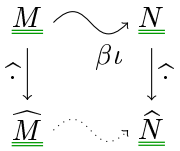
$$\widehat{\underline{x}} = x \qquad \widehat{\lambda^{\psi}\underline{x}_1.\underline{M}_2} = \lambda x.\widehat{\underline{M}}_2 \qquad \widehat{\underline{M}_1 @^{\psi} \underline{N}_2} = \widehat{\underline{M}}_1\ \widehat{\underline{N}}_2 \qquad \widehat{\underline{\mathtt{Unit}}} = \mathtt{Unit}$$

$$\widehat{\varnothing} = \varnothing \qquad \widehat{\Sigma, \underline{\alpha}} = \widehat{\Sigma}, \alpha \qquad \widehat{\Gamma, (\underline{x}_1 : \underline{\tau}_2)} = \widehat{\Gamma}, (\widehat{\underline{x}}_1 : \widehat{\underline{\tau}}_2) \qquad \widehat{\underline{x}_1 : \underline{\tau}_2} = (\widehat{\underline{x}}_1 : \widehat{\underline{\tau}}_2)$$

$$\widehat{\Delta, (h : \underline{x}_1 : \underline{\tau}_2)} = \widehat{\Delta}, (\widehat{\underline{x}}_1 : \widehat{\underline{\tau}}_2)$$

$$\widehat{\underline{\alpha}} = \alpha \qquad \widehat{\underline{\sigma}_1 \to \underline{\tau}_2} = \widehat{\underline{\sigma}}_1 \to \widehat{\underline{\tau}}_2 \qquad \widehat{\forall \underline{\alpha}_1.\underline{\tau}_2} = \forall \alpha.\widehat{\underline{\tau}}_3 \qquad \widehat{\underline{\mathtt{Unit}}} = \mathtt{Unit}$$

Lemma 1 (Forward simulation with F)**.** Proof page 21
We have the following commutative diagram:



Lemma 2 (Translation to F)**.** Proof page 21
If $\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash \underline{M} : \underline{\tau}$ *holds, then* $\widehat{\Sigma}, \widehat{\Gamma}, \widehat{\mathrm{Z}}, \widehat{\Delta} \vdash_{\mathsf{F}} \widehat{\underline{M}} : \widehat{\underline{\tau}}$ *holds.*

Lemma 3 (Normalization)**.** *If* $\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash \underline{M} : \underline{\tau}$ *holds, then* $\underline{M}$ *strongly normalizes.*

Proof. With Lemma 1, Lemma 2 and strong normalization in System F. □

3.3 Soundness

We show that $\mathsf{F}_{\lambda\eta}$ is sound with the subject reduction and progress lemmas, respectively Lemma 11 and Lemma 12. As usual, to show subject reduction, we need a few lemmas about substitution. We have one lemma for each environment holding term variables. But first we have a lemma, judgment substitution, about free type variables which is essential for coercion substitution in the presence of the restriction we used. This lemma was false in [2], leading to an unsound type system.

Short proofs are just below their lemmas, while more technical ones are defered until Section B.

Lemma 4 (Judgment substitution)**.** Proof page 21
If $\Sigma, \underline{\alpha}; \Gamma; \mathrm{Z}; \Delta \vdash \underline{M} : \underline{\tau}$ *holds, then* $(\Sigma, \underline{\alpha}; \Gamma; \mathrm{Z}; \Delta \vdash \underline{M} : \underline{\tau})[\underline{\alpha} \leftarrow \underline{\sigma}]$ *holds.*

Lemma 5 (Substitution)**.** Proof page 21
If $\Sigma; \Gamma; \varnothing; \varnothing \vdash \underline{N} : \underline{\tau}$ *and* $\Sigma; \Gamma, (\underline{x} : \underline{\tau}); \varnothing; \varnothing \vdash \underline{M} : \underline{\sigma}$ *hold, then* $\Sigma; \Gamma; \varnothing; \varnothing \vdash \underline{M}[\underline{x} \leftarrow \underline{N}] : \underline{\sigma}$ *holds.*

Lemma 6 is expressed in a quite unusual way, because of the restriction we added in CAbs to preserve the coercion-erasure semantics. This comes from the fact, that in the proof of subject reduction, when inversing the typing relation in the CIota reduction rule case, a TApp typing rule appears to get an arrow for the application.

Lemma 6 (Coercion substitution)**.** *If* $(\Sigma, \underline{\alpha}; \Gamma; \varnothing; \varnothing \vdash \underline{N} : \underline{\tau} \to \underline{\alpha})[\underline{\alpha} \leftarrow \underline{\tau'}]$ *and* $\Sigma, \underline{\alpha}; \Gamma, (\underline{x} : \underline{\tau} \to \underline{\alpha}); \mathrm{Z}; \Delta \vdash \underline{M} : \underline{\sigma}$ *hold, then* $(\Sigma, \underline{\alpha}; \Gamma; \mathrm{Z}; \Delta \vdash \underline{M}[\underline{x} \leftarrow \underline{N}] : \underline{\sigma})[\underline{\alpha} \leftarrow \underline{\tau'}]$ *holds.*

Proof. Applying Lemma 5, then Lemma 4. □

What is interesting in the following lemmas, is the play of the environments. We keep in mind that $\mathrm{Z} = \underline{x} : \underline{\tau}$ is used to type an erasable wrapping of $\underline{x}$, and that Δ is a resource that we use as a stack.

Lemma 7. Proof page 23
If $\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash \underline{N} : \underline{\tau}$ *and* $\Sigma; \Gamma; \underline{x} : \underline{\tau}; \varnothing \vdash \underline{M} : \underline{\sigma}$ *hold, then* $\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash \underline{M}[\underline{x} \leftarrow \underline{N}] : \underline{\sigma}$ *holds.*

Lemma 8. Proof page 23
If $\Sigma; \Gamma; \varnothing; \varnothing \vdash \underline{N} : \underline{\tau}$ *and* $\Sigma; \Gamma; \varnothing; (h : \underline{x} : \underline{\tau}) \vdash \underline{M} : \underline{\sigma}$ *hold, then* $\Sigma; \Gamma; \varnothing; \varnothing \vdash \underline{M}[h \leftarrow \square][\underline{x} \leftarrow \underline{N}] : \underline{\sigma}$ *holds.*

Lemma 9. Proof page 23
If $\Sigma; \Gamma; \varnothing; (h : \underline{y} : \underline{\tau}) \vdash \underline{E}\left[(\lambda^{\square}\underline{x}.\underline{M})@^{h}\underline{N}\right] : \underline{\sigma}$ *holds, then* $\Sigma; \Gamma, (\underline{y} : \underline{\tau}); \varnothing; \varnothing \vdash \underline{E}\left[\underline{M}[\underline{x} \leftarrow \underline{N}]\right] : \underline{\sigma}$ *holds.*

Lemma 10. Proof page 23
If $\Sigma; \Gamma; \mathrm{Z}; \Delta, (h : \underline{y} : \underline{\tau}) \vdash \underline{M} : \underline{\sigma}$ *and* $\Sigma; \Gamma; \underline{x} : \underline{\tau'}; \varnothing \vdash \underline{N} : \underline{\tau}$ *hold, then* $\Sigma; \Gamma; \mathrm{Z}; \Delta, (h' : \underline{x} : \underline{\tau'}) \vdash \underline{M}[\underline{y} \leftarrow \underline{N}][h \leftarrow h'] : \underline{\sigma}$ *holds.*

In subject reduction, we use the notation $\rightsquigarrow_{\beta\iota}$ to talk about the union of $\rightsquigarrow_{\beta}$ and $\rightsquigarrow_{\iota}$. So it states that for any well typed term in any environments, if it reduces in any way, then it is typed in the same environments with the same type. And progress tells that any well typed term in any environments is either a value, or reduces in some way.

Lemma 11 (Subject Reduction)**.** Proof page 23
If $\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash \underline{\underline{M}} : \underline{\underline{\tau}}$ *and* $\underline{\underline{M}} \rightsquigarrow_{\beta\iota} \underline{\underline{N}}$ *hold, then* $\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash \underline{\underline{N}} : \underline{\underline{\tau}}$ *holds.*

Lemma 12 (Progress)**.** Proof page 23
If $\Sigma; \Gamma; \mathrm{Z}; \varnothing \vdash \underline{\underline{M}} : \underline{\underline{\tau}}$ *holds, then either* $\underline{\underline{M}}$ *is a variable or* $\underline{\underline{M}} \rightsquigarrow_{\beta\iota} \underline{\underline{N}}$ *holds.*

4 Properties

There are three main results about $\mathsf{F}_{\lambda\eta}$. On the one hand, we show that both F_{η} and $\mathsf{xML}^{\mathsf{F}}$ are included into $\mathsf{F}_{\lambda\eta}$ in Sections 4.1 and 4.2. On the other hand, we show that we preserve the coercion-erasure semantics in Section 4.3.

To show an inclusion, we need to show that the translation of a well typed term of the source language is well typed in the target language, and that both terms erases to the same Lambda Calculus term.

Figure 9: Translation of F_η

$$\alpha^\star = \alpha \qquad (\tau \to \sigma)^\star = \tau^\star \to \sigma^\star \qquad (\forall \alpha.\tau)^\star = \forall \alpha^\star.\tau^\star$$

Definition of $(\sigma_\eta \subseteq \tau_\eta)^\star$

$$(\text{sub})^\star = \lambda^\square z.z \qquad (\text{dist})^\star = \lambda^\square z.\lambda^h x.z@^h x \qquad (\text{congruence})(c)^\star = \lambda^\square z.c^\star@^\square x$$

$$(\text{arrow})(c_1, c_2)^\star = \lambda^\square z.\lambda^h x.c_2{}^\star@^\square(z@^h(c_1{}^\star@^\square x)) \qquad (\text{trans})(c_1, c_2)^\star = \lambda^\square z.c_2{}^\star@^\square(c_1{}^\star@^\square x)$$

Definition of $(\Gamma_\eta \vdash_\eta M_\eta : \tau_\eta)^\star$

$$(\text{var})^\star = x \qquad (\text{add hyp})(m)^\star = m^\star \qquad (\to I_\forall)(m)^\star = \lambda^\square x.m^\star$$

$$(\to E_\forall)(m, n)^\star = m^\star@^\square n^\star \qquad (\text{cont})(m, c)^\star = c^\star@^\square m^\star$$

$$\varnothing^\star = \varnothing \qquad (\Gamma_\eta, x : \sigma_\eta)^\star = \Gamma_\eta{}^\star, (x : \sigma_\eta{}^\star)$$

4.1 Inclusion of F_η

We define the translation from F_η to $\mathsf{F}_{\lambda\eta}$ in Figure 9. This translation is done in several steps. First, we have a function from F_η types to $\mathsf{F}_{\lambda\eta}$ types. Then, we have a function from F_η containment judgments to $\mathsf{F}_{\lambda\eta}$ coercions. This function matches on the nodes of the derivation tree. When the tree is not a leaf, its subtrees are written as arguments of the derivation rule name (e.g. $(\text{arrow})(c_1, c_2)$). We do the same for the function between judgments and $\mathsf{F}_{\lambda\eta}$ normal terms. And we finally have a translation of F_η environments. We can easily see that the translation of a well typed F_η term drops on the initial F_η term.

Using these definitions, we show that F_η is included in $\mathsf{F}_{\lambda\eta}$ in Lemma 13.

Lemma 13 (Inclusion of F_η)**.** Proof page 24
The following assertion holds:

- Containment*: If* $\sigma_\eta \subseteq \tau_\eta$*,* $\Sigma \vdash \Gamma$*,* $\Sigma \vdash \sigma_\eta{}^\star$*, and* $\Sigma \vdash \tau_\eta{}^\star$ *hold, then* $\Sigma; \Gamma; \varnothing; \varnothing \vdash (\sigma_\eta \subseteq \tau_\eta)^\star : \sigma_\eta{}^\star \to \tau_\eta{}^\star$ *holds.*
- Typing*: If* $\Gamma_\eta \vdash_\eta M_\eta : \sigma_\eta$*,* $\Sigma \vdash \Gamma_\eta{}^\star$*, and* $\Sigma \vdash \sigma_\eta{}^\star$ *hold, then* $\Sigma; \Gamma_\eta{}^\star; \varnothing; \varnothing \vdash (\Gamma_\eta \vdash_\eta M_\eta : \sigma_\eta)^\star : \sigma_\eta{}^\star$ *holds.*

4.2 Inclusion of xML^F

We define the translation from xML^F to $\mathsf{F}_{\lambda\eta}$ in Figure 10. We see again that the translation of a well typed xML^F term, drops on the same Lambda Calculus term than the initial xML^F term. The drop function on xML^F terms is defined in a natural way.

Lemma 14 (Inclusion of xML^F)**.** Proof page 24
The following assertions hold:

- Instantiation*: If* $\Gamma_\mathsf{x} \vdash_\mathsf{x} \phi_\mathsf{x} : \sigma_\mathsf{x} \leq \tau_\mathsf{x}$, $\Sigma \vdash \Gamma_\mathsf{x}^\circ$, $\Sigma \vdash \sigma_\mathsf{x}^\circ$, *and* $\Sigma \vdash \tau_\mathsf{x}^\circ$ *hold, then* $\Sigma; \Gamma_\mathsf{x}^\circ; \varnothing; \varnothing \vdash \phi_\mathsf{x}^\circ : (\tau_\mathsf{x} \geq \sigma_\mathsf{x})^\circ$ *holds.*
- Typing*: If* $\Gamma_\mathsf{x} \vdash_\mathsf{x} M_\mathsf{x} : \tau_\mathsf{x}$, $\Sigma \vdash \Gamma_\mathsf{x}^\circ$, *and* $\Sigma \vdash \tau_\mathsf{x}^\circ$ *hold, then* $\Sigma; \Gamma^\circ; \varnothing; \varnothing \vdash M_\mathsf{x}^\circ : \tau_\mathsf{x}^\circ$ *holds.*

4.3 Bisimulation

We need a few lemmas showing the invariants of the type system. Lemma 15 tells that when $\underline{M}$ is typed with $\mathrm{Z} = \underline{x} : \underline{\tau}$ then $\underline{M}$ is just an erasable wrapping of $\underline{x}$. This comes from the idea that coercions behave like the identity function. Lemma 16 mainly tells the same sort of result. If something is typable with a Δ ending with $(h' : \underline{x} : \underline{\tau})$ then it's an erasable wrapping of the only application node flashed with a h'.

Lemma 15. Proof page 24
If $\Sigma; \Gamma; \underline{x} : \underline{\tau}; \Delta \vdash \underline{M} : \underline{\sigma}$ *holds, then* $\lfloor \underline{M} \rfloor = x$ *holds.*

Lemma 16. Proof page 24
If $\Sigma; \Gamma; \mathrm{Z}; \Delta, (h' : \underline{x} : \underline{\tau}) \vdash \underline{E}\left[\underline{M} @^h \underline{N}\right] : \underline{\sigma}$ *holds, then* $\lfloor \underline{E}[\cdot] \rfloor = \lfloor \cdot \rfloor$ *holds.*

To show the bisimulation, we need an additional hypothesis $\Sigma \vdash_{bi} \Gamma$ on the environments that asks coercion variables to be parametric on their return type. We did not use this judgment in the type system because it breaks Lemma 4 and would make the proof of Lemma 6 not modular.

$$\textsc{EnvEmpty}\ \ \Sigma \vdash_{bi} \varnothing \qquad\qquad \textsc{EnvTerm}\ \ \frac{\Sigma \vdash_{bi} \Gamma \qquad \Sigma \vdash \underline{\tau} \qquad x \notin \mathrm{fv}(\Gamma)}{\Sigma \vdash_{bi} \Gamma, (\underline{x} : \underline{\tau})}$$

$$\textsc{EnvCoercion}\ \ \frac{\Sigma \vdash_{bi} \Gamma \qquad \Sigma \vdash \underline{\tau}_1 \qquad \Sigma \vdash \underline{\sigma}_2 \qquad x \notin \mathrm{fv}(\Gamma)}{\Sigma \vdash_{bi} \Gamma, (x : \underline{\tau}_1 \to \underline{\sigma}_2)}$$

Lemma 17. Proof page 24
If $\Sigma \vdash_{bi} \Gamma$, $\Sigma; \Gamma; \varnothing; \varnothing \vdash \underline{M} : \underline{\tau}$, $\lfloor \underline{M} \rfloor = (\lambda x.M_\lambda)\ N_\lambda$, *and* $\underline{M} \not\rightsquigarrow_\iota$ *hold, then* $\underline{M} \rightsquigarrow_\beta \underline{N}$ *and* $\lfloor \underline{N} \rfloor = M_\lambda[x \leftarrow N_\lambda]$ *hold.*

Lemma 18 (Bisimulation)**.** Proof page 25
The following assertions hold:

(1) If $\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash \underline{M} : \underline{\tau}$ *and* $\underline{M} \rightsquigarrow_\beta \underline{N}$ *hold, then* $\lfloor \underline{M} \rfloor \rightsquigarrow \lfloor \underline{N} \rfloor$ *holds.*

Figure 10: Translation of $\mathsf{xML}^{\mathsf{F}}$

$$\varnothing^\circ = \varnothing \qquad (\Gamma_{\mathsf{x}}, x : \tau_{\mathsf{x}})^\circ = \Gamma_{\mathsf{x}}{}^\circ, (\underline{x} : \tau_{\mathsf{x}}{}^\circ) \qquad (\Gamma_{\mathsf{x}}, \alpha \geq \tau_{\mathsf{x}})^\circ = \Gamma_{\mathsf{x}}{}^\circ, (\underline{i_\alpha} : (\alpha \geq \tau_{\mathsf{x}})^\circ)$$

Definition of $M_{\mathsf{x}}{}^\circ$

$$x^\circ = \underline{x} \qquad (\lambda(x : \tau_{\mathsf{x}})M_{\mathsf{x}})^\circ = \underline{\lambda^\square \underline{x}.M_{\mathsf{x}}{}^\circ} \qquad (M_{\mathsf{x}}\, N_{\mathsf{x}})^\circ = \underline{M_{\mathsf{x}}{}^\circ @^\square N_{\mathsf{x}}{}^\circ}$$

$$(\Lambda(\alpha \geq \tau_{\mathsf{x}})M_{\mathsf{x}})^\circ = \underline{\lambda^\square \underline{i_\alpha}.M_{\mathsf{x}}{}^\circ} \qquad (M_{\mathsf{x}}\, \phi_{\mathsf{x}})^\circ = \underline{\phi_{\mathsf{x}}{}^\circ @^\square M_{\mathsf{x}}{}^\circ}$$

$$(\mathtt{let}\, x = M_{\mathsf{x}}\, \mathtt{in}\, N_{\mathsf{x}})^\circ = \underline{(\lambda^\square \underline{x}.N_{\mathsf{x}}{}^\circ) @^\square M_{\mathsf{x}}{}^\circ}$$

Definition of $\phi_{\mathsf{x}}{}^\circ$

$$\tau_{\mathsf{x}}{}^\circ = \underline{\lambda^\square \underline{z}.\underline{z}} \qquad (\phi_{\mathsf{x}}; \psi_{\mathsf{x}})^\circ = \underline{\lambda^\square \underline{z}.\psi_{\mathsf{x}}{}^\circ @^\square (\phi_{\mathsf{x}}{}^\circ @^\square \underline{z})} \qquad \mathbf{1}^\circ = \underline{\lambda^\square \underline{z}.\underline{z}} \qquad \&^\circ = \underline{\lambda^\square \underline{z}.\underline{z} @^\square (\lambda^\square \underline{y}.\underline{y})}$$

$$\parr^\circ = \underline{\lambda^\square \underline{z}.\lambda^\square \underline{i_\alpha}.\underline{z}} \qquad (!\alpha)^\circ = \underline{i_\alpha} \qquad (\forall(\geq \phi_{\mathsf{x}}))^\circ = \underline{\lambda^\square \underline{z}.\lambda^\square \underline{i_\alpha}.\underline{z} @^\square (\lambda^\square \underline{y}.\underline{i_\alpha} @^\square (\phi_{\mathsf{x}}{}^\circ @^\square \underline{y}))}$$

$$(\forall(\alpha \geq)\phi_{\mathsf{x}})^\circ = \underline{\lambda^\square \underline{z}.\lambda^\square \underline{i_\alpha}.\phi_{\mathsf{x}}{}^\circ @^\square (\underline{z} @^\square \underline{i_\alpha})}$$

Definition of $\tau_{\mathsf{x}}{}^\circ$

$$\alpha^\circ = \underline{\alpha} \qquad (\sigma_{\mathsf{x}} \to \tau_{\mathsf{x}})^\circ = \underline{\sigma_{\mathsf{x}}{}^\circ \to \tau_{\mathsf{x}}{}^\circ} \qquad \bot^\circ = \underline{\forall \underline{\alpha}.\underline{\alpha}} \qquad (\forall(\alpha \geq \sigma_{\mathsf{x}})\tau_{\mathsf{x}})^\circ = \underline{\forall \underline{\alpha}.(\alpha \geq \sigma_{\mathsf{x}})^\circ \to \tau_{\mathsf{x}}{}^\circ}$$

$$(\tau_{\mathsf{x}} \geq \sigma_{\mathsf{x}})^\circ = \underline{\sigma_{\mathsf{x}}{}^\circ \to \tau_{\mathsf{x}}{}^\circ}$$

(2) If $\Sigma; \Gamma; Z; \Delta \vdash \underline{\underline{M}} : \underline{\tau}$ *and* $\underline{\underline{M}} \leadsto_\iota \underline{N}$ *hold, then* $\lfloor \underline{\underline{M}} \rfloor = \lfloor \underline{N} \rfloor$ *holds.*

(3) If $\Sigma \vdash_{bi} \Gamma$, $\Sigma; \Gamma; Z; \Delta \vdash \underline{M} : \underline{\tau}$, *and* $\lfloor \underline{M} \rfloor \leadsto M_\lambda$ *hold, then* $\underline{M} \leadsto_\iota^\star \leadsto_\beta \underline{N}$ *and* $\lfloor \underline{N} \rfloor = M_\lambda$ *hold.*

5 Conclusion and Future Work

We designed $\mathsf{F}_{\lambda\eta}$, a language with coercions containing both F_η and xML^F, and preserving the coercion-erasure semantics. To do so, we added restrictions to disallow some interactions between contra-variance and abstraction. We can only write abstractions which are parametric on the return type of their coercion. However, we can weaken this restriction without much effort and ask for parametricity either on the right side of the arrow (the return type), or on the left side (the argument type). It should be also possible to completely remove this restriction, but it would imply to add new non trivial reduction rules, which is still under investigation.

We currently only offer normal term abstraction, coercion abstraction returning a term, coercion between terms, and η-expansion. But we could authorize any kind of abstraction or coercion. For example, we could add coercion abstraction returning a coercion as in

$$\lambda^\square c_1.\lambda^\square c_2.\lambda^\square f.\lambda^h x.c_2 @^\square (\underline{f} @^h (\underline{c_1} @^\square \underline{x}))$$

which is a function taking two coercions $\underline{c_1}$ and $\underline{c_2}$ and building a coercion on functions using $\underline{c_1}$ to coerce the argument and $\underline{c_2}$ for the result. We could also add coercions between coercions, or even term abstraction returning a coercion.

We can also extend one of the idea in [1], which is describing the language using containment judgment instead of terms. We would write coercions using proof terms of a containment judgment instead of lambda calculus. In this case it would be interesting to see if there is a way to deduce a coercion from its type, and see if its normal form is unique. And we could look for an equivalence proof between the current lambda description and this containment version.

Finally, we can show that one step in xML^F corresponds to at least one step in $\mathsf{F}_{\lambda\eta}$. Thus, we obtain strong normalization for xML^F.

References

[1] John C. Mitchell. **Polymorphic type inference and containment**. *Information and Computation*, 76, 211-249, 1988.

[2] Giulio Manzonetto and Paolo Tranquilli. **Harnessing ML$^\mathsf{F}$ with the Power of System F**. *MFCS 2010*, 525-536.

[3] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. **System F with Type Equality Coercions**. *TLDI'07*, January 2007.

Figure 11: Definition of the Lambda Calculus

$$
\begin{array}{rr}
x, y & \text{variables} \\
M, N ::= x \mid \lambda x.M \mid M\ N \mid \texttt{Unit} & \text{terms} \\
E[\cdot] ::= \cdot \mid \lambda x.E[\cdot] \mid E[\cdot]\ M \mid M\ E[\cdot] & \text{contexts}
\end{array}
$$

$$
\textsc{Context}\ \frac{M \rightsquigarrow M}{E[M] \rightsquigarrow E[M]} \qquad \textsc{Beta}\ (\lambda x.M)\ N \rightsquigarrow M[x \leftarrow N]
$$

[4] Didier Le Botlan and Didier Rémy. **ML$^\mathsf{F}$: Raising ML to the power of System F**. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 27-38, August 2003.

[5] Didier Rémy and Boris Yakobowski. **A Church-Style Intermediate Language for ML$^\mathsf{F}$**. In *Functional and Logic Programming*, volume 6009 of *Lecture Notes in Computer Science*, pages 24-39, Springer Berlin / Heidelberg, 2010.

[6] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. **Inheritance as implicit coercion**. In *Information and Computation*, 93(1):172-221, July 1991.

A Figures

B Technical Proofs

Proof of Lemma 1. By induction on $\underline{\underline{M}} \rightsquigarrow_{\beta\iota} \underline{\underline{N}}$. Then it's always β-reduction, but for the context rules which are handled with the context rule of System F and the induction hypothesis, and the NIotaUp rule which is handled with the context rule and the β-reduction rule. □

Proof of Lemma 2. By induction on $\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash \underline{\underline{M}} : \underline{\tau}$. It suffices to see that the erasure of $\mathsf{F}_{\lambda\eta}$ typing rules are included in those of System F. We just need to use the weakening property in System F for CApp, LApp, and NApp. And we need to use TAbs in addition to Abs to translate CApp. □

Proof of Lemma 4. By induction on $\Sigma, \underline{\alpha}; \Gamma; \mathrm{Z}; \Delta \vdash \underline{\underline{M}} : \underline{\tau}$. In addition to the type substitution along the whole tree, we update each leaf $\Sigma, \underline{\alpha}, \Sigma' \vdash \underline{\alpha}$ with a derivation tree of $\Sigma, \mathrm{ftv}(\underline{\sigma}), \Sigma' \vdash \underline{\sigma}$ which is possible, because the free type variables of $\underline{\sigma}$ are in the new type environment. Note that the Σ and Γ environments can only grow from the root to the leaves. This is why we can write $\Sigma, \underline{\alpha}, \Sigma'$. □

Proof of Lemma 5. By induction on $\Sigma; \Gamma, (\underline{\underline{x}} : \underline{\tau}); \varnothing; \varnothing \vdash \underline{\underline{M}} : \underline{\sigma}$. In addition to the term substitution along the whole tree, we update each leaf $\Sigma, \Sigma'; \Gamma, \Gamma'; \varnothing; \varnothing \vdash \underline{\underline{x}} : \underline{\tau}$ with a derivation

Figure 12: Definition of F

$$
\begin{array}{rr}
x, y & \text{term variables} \\
\alpha, \beta & \text{type variables} \\
M, N ::= x \mid \lambda x.M \mid M\ N \mid \texttt{Unit} & \text{terms} \\
p ::= x \mid p\ v & \text{prevalues} \\
v ::= p \mid \lambda x.v \mid \texttt{Unit} & \text{values} \\
\sigma, \tau ::= \alpha \mid \tau \to \sigma \mid \forall \alpha.\tau \mid \texttt{Unit} & \text{types} \\
E[\cdot] ::= \cdot \mid \lambda x.E[\cdot] \mid E[\cdot]\ M \mid M\ E[\cdot] & \text{contexts} \\
\Gamma ::= \varnothing \mid \Gamma, (x : \tau) \mid \Gamma, \alpha & \text{environments} \\
\Gamma \vdash_{\mathsf{F}} M : \tau & \text{term judgments} \\
\Gamma \vdash_{\mathsf{F}} \tau & \text{type judgments} \\
\Gamma \vdash_{\mathsf{F}} ok & \text{environment judgments}
\end{array}
$$

$$\textsc{EnvEmpty}\ \varnothing \vdash_{\mathsf{F}} ok$$

$$\textsc{EnvTerm}\ \frac{\Gamma \vdash_{\mathsf{F}} \tau \qquad x \notin dom(\Gamma)}{\Gamma, (x : \tau) \vdash_{\mathsf{F}} ok}$$

$$\textsc{EnvType}\ \frac{\Gamma \vdash_{\mathsf{F}} ok \qquad \alpha \notin \Gamma}{\Gamma, \alpha \vdash_{\mathsf{F}} ok}$$

$$\textsc{TypeVariable}\ \frac{\Gamma \vdash_{\mathsf{F}} ok \qquad \alpha \in \Gamma}{\Gamma \vdash_{\mathsf{F}} \alpha}$$

$$\textsc{TypeArrow}\ \frac{\Gamma \vdash_{\mathsf{F}} \tau \qquad \Gamma \vdash_{\mathsf{F}} \sigma}{\Gamma \vdash_{\mathsf{F}} \tau \to \sigma}$$

$$\textsc{TypeForall}\ \frac{\Gamma, \alpha \vdash_{\mathsf{F}} \tau}{\Gamma \vdash_{\mathsf{F}} \forall \alpha.\tau}$$

$$\textsc{TypeUnit}\ \frac{\Gamma \vdash_{\mathsf{F}} ok}{\Gamma \vdash_{\mathsf{F}} \texttt{Unit}}$$

$$\textsc{Ax}\ \frac{\Gamma, (x : \tau) \vdash_{\mathsf{F}} ok}{\Gamma, (x : \tau) \vdash_{\mathsf{F}} x : \tau}$$

$$\textsc{Abs}\ \frac{\Gamma, (x : \tau) \vdash_{\mathsf{F}} M : \sigma}{\Gamma \vdash_{\mathsf{F}} \lambda x.M : \tau \to \sigma}$$

$$\textsc{App}\ \frac{\Gamma \vdash_{\mathsf{F}} M : \tau \to \sigma \qquad \Gamma \vdash_{\mathsf{F}} N : \tau}{\Gamma \vdash_{\mathsf{F}} M\ N : \sigma}$$

$$\textsc{TAbs}\ \frac{\Gamma, \alpha \vdash_{\mathsf{F}} M : \tau}{\Gamma \vdash_{\mathsf{F}} M : \forall \alpha.\tau}$$

$$\textsc{TApp}\ \frac{\Gamma \vdash_{\mathsf{F}} M : \forall \alpha.\tau \qquad \Gamma \vdash_{\mathsf{F}} \sigma}{\Gamma \vdash_{\mathsf{F}} M : \tau[\alpha \leftarrow \sigma]}$$

tree of $\Sigma, \Sigma'; \Gamma, \Gamma'; \varnothing; \varnothing \vdash \underline{\underline{N}} : \underline{\underline{\tau}}$ using the derivation of $\Sigma; \Gamma; \varnothing; \varnothing \vdash \underline{\underline{N}} : \underline{\underline{\tau}}$ in hypothesis and the derivation of $\Sigma, \Sigma'; \Gamma, (\underline{\underline{x}} : \underline{\underline{\tau}}), \Gamma'; \varnothing; \varnothing \vdash \underline{\underline{x}} : \underline{\underline{\tau}}$ for the part about Γ'. □

Proof of Lemma 7. By induction on $\Sigma; \Gamma; \underline{x} : \underline{\tau}; \varnothing \vdash \underline{M} : \underline{\sigma}$. We modify the LAx leaves with $\Sigma; \Gamma; \mathrm{Z}; \Delta \vdash \underline{N} : \underline{\tau}$ where we adapted the Σ and Γ environments. And along the whole tree we add Z and Δ, because every rule with a Z allows a Δ. □

Proof of Lemma 8. By induction on $\Sigma; \Gamma; \varnothing; (h : \underline{x} : \underline{\tau}) \vdash \underline{M} : \underline{\sigma}$. The only intersting case is NApp. Because of the flash substitution, the new rule is App, so we have to check that we can produce the premises using Lemma 7. Because Z and Δ are equal to $\varnothing$, the left premise is fine as it is. For the right premise, we use Lemma 7, and we are fine because x does not appear in M. □

Proof of Lemma 9. By induction on $\Sigma; \Gamma; \varnothing; (h : \underline{y} : \underline{\tau}) \vdash E\left[\underline{\underline{(\lambda^{\square} \underline{x}.\underline{M})@^h N}}\right] : \underline{\sigma}$, and then inversion of $E[\cdot]$. The intersting case is NApp when $E[\cdot] = \cdot$. We use Lemma 5 with $\Sigma; \Gamma, (\underline{y} : \underline{\tau}), (\underline{x} : \underline{\tau'}); \varnothing; \varnothing \vdash \underline{M} : \underline{\sigma}$ and $\Sigma; \Gamma, (\underline{y} : \underline{\tau}); \varnothing; \varnothing \vdash \underline{N} : \underline{\tau'}$. We get the first judgment from the inversion of the typing rule. The second judgment also results from $\Sigma; \Gamma; \underline{y} : \underline{\tau}; \varnothing \vdash \underline{N} : \underline{\tau'}$ which comes from the inversion of the typing rule. □

Proof of Lemma 10. By induction on $\Sigma; \Gamma; \mathrm{Z}; \Delta, (h : \underline{y} : \underline{\tau}) \vdash \underline{M} : \underline{\sigma}$. The only intersting case is NApp. To rebuild the rule, we do an induction on the right premise, because this is where the term substitution takes place. □

Proof of Lemma 11. By induction on $\underline{\underline{M}} \rightsquigarrow_{\beta\iota} \underline{\underline{N}}$.

- ContextBeta and ContextIota: By induction on $E[\cdot]$, by inversion of the typing relation and using both induction hypotheses.
- Beta: By inversion of the typing relation and Lemma 5.
- CIota: By inversion of the typing relation (there is a TApp involved), some rewriting of the judgments and Lemma 6.
- LIota: By inversion of the typing relation and Lemma 7.
- NIotaDown: There are two cases depending on ψ. If it is $\square$, then by inversion of the typing relation and Lemma 8 we have the result. If it s h', then by inversion of the typing relation and Lemma 10 we have the result.
- NIotaUp: By inversion of the typing relation and Lemma 9.

□

Proof of Lemma 12. We show that if $\Sigma; \Gamma; \mathrm{Z}; \varnothing \vdash E\left[\underline{\underline{M}}\right] : \underline{\underline{\tau}}$ holds, then either $E\left[\underline{\underline{M}}\right] \rightsquigarrow_{\beta\iota} E\left[\underline{\underline{N}}\right]$ or $\underline{\underline{M}} = \underline{\underline{v}}$ holds. By induction on $\underline{\underline{M}}$. Almost all the cases are solved with ContextBeta or ContextIota. The remaining cases are $\underline{\underline{\underline{\underline{v_2}}@^{\psi}\underline{\underline{v'}}}_3}_1$.

- $\underline{\underline{\underline{\underline{p_2}}@^{\psi}\underline{\underline{v}}}_3}_1$: This is a value.

- $(\lambda^{\psi'} x_3.v_4)@^{\psi} v'_5$: By cases and by typing, we have either
  - $(\lambda^{\psi} x.v)@^{\square} v'$ and Beta or NIotaDown apply, or
  - $(\lambda^{\square} x.v)@^{\square} v'$ and CIota applies, or
  - $(\lambda^{\square} x.v)@^{\square} v'$ and LIota applies, or
  - $(\lambda^{h'} x.v)@^{h} v'$ and NIotaDown applies.
  - $(\lambda^{\square} x.v)@^{h} v'$ and by typing we can go up the context $E[\cdot]$ until we find the lambda node associated with h and NIotaUp applies.
- $\texttt{Unit}_2@^{\psi} v_3$: Not typeable.

□

Proof of Lemma 13. Containment: By induction on $\sigma \subseteq \tau$.
Typing: By induction on $\Gamma \vdash_{\eta} M : \sigma$.

- (var): Ax.
- (add hyp): By induction.
- $(\to I_{\forall})$: TAbs* and Abs.
- $(\to E_{\forall})$: TAbs*, App, and TApp*.
- (cont): LApp.

□

Proof of Lemma 14. Instantiation: By induction on $\Gamma \vdash_{\mathsf{x}} \phi : \sigma \leq \tau$.
Typing: By induction on $\Gamma \vdash_{\mathsf{x}} M : \tau$. □

Proof of Lemma 15. By induction on $\Sigma; \Gamma; x : \tau; \Delta \vdash M : \sigma$. For each case with a Z which is not LAx, we look which premise contains the Z and we verify that it is the part kept by the drop function. □

Proof of Lemma 16. By induction on $E[\cdot]$. All the cases are obvious but $\Sigma; \Gamma; \mathrm{Z}; \Delta, (h : x : \tau) \vdash \lambda^{h'} x.E\left[M@^{h}N\right] : \sigma$. By typing, we have $E[\cdot] = E'[E''[\cdot]@^{h'} N']$. We apply the induction hypothesis for $E'[\cdot]$ and $E''[\cdot]$. □

Proof of Lemma 17. Let's consider the following forms $E[\lambda^{\square} x.M]@^{\square} N$ with $\lfloor E[\cdot] \rfloor = \lfloor \cdot \rfloor$. We have $\lfloor E[\lambda^{\square} x.M]@^{\square} N \rfloor = (\lambda x.\lfloor M \rfloor)\ \lfloor N \rfloor$. By typing and the fact that we are in ι-normal form, we show by induction on $E[\cdot]$ that $E[\cdot] = \cdot$.

- $\lambda^{\square} x.E[M]$: By typing of the application.
- $\lambda^{h} x.E[M]$ with h present in $E[M]$: We can do a ι-reduction.
- $M @^{\square} E[N]$: Either M is a variable and it's not well typed, or it's a lambda and we can do a ι-reduction.
- $E[M] @^{\square} N$: By typing we have $\lambda^{\psi} x.M$ on the left.
- $E[M] @^{h} N$: We can do a ι-reduction.

□

Proof of Lemma 18. (1) By induction on $M \rightsquigarrow_{\beta} N$.

- ContextBeta: The context cannot go through red.
- Beta: Drops on a β-reduction.

(2) By induction on $M \rightsquigarrow_{\iota} N$.

- ContextIota: Induction hypothesis.
- CIota: Ok.
- LIota: With Lemma 15, we have $\lfloor M \rfloor = x$.
- NIotaDown: We use Lemma 16 and Lemma 15.
- NIotaUp: We use Lemma 16 and Lemma 15.

(3) With Lemma 17 and Lemma 3.

□