

A generalization of F_η with abstraction over retyping functions

Julien Cretin
supervised by Didier Rémy
(Gallium, INRIA)

December 3, 2010

Motivation

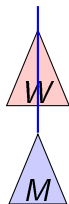
Coercions allow to change types a posteriori with no run-time cost. We can use them:

- ▶ to describe subtyping ($F_{<}$: [Breazu-Tannen, Coquand, Gunter, and Scedrov '91])
- ▶ to model GADTs or type families (F_C [Sulzmann, Chakravarty, Peyton Jones, and Donnelly '07]),
- ▶ to do deep instantiations (F_η [Mitchell '88]),
- ▶ to extend ML with first-class polymorphism (ML^F [Le Botlan and Rémy '03]), or more simply
- ▶ to talk about instantiation and generalization in any Church-style language (for instance F).

Handling coercions

There are several ways of handling coercions.

- ▶ As erasable contexts: coercions appear around terms.

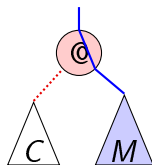
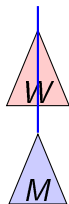


Erasable context only

Handling coercions

There are several ways of handling coercions.

- ▶ As erasable contexts: coercions appear around terms.
- ▶ As functions: coercions apply on terms.

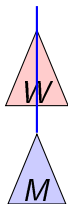


Erasable context only Functions only

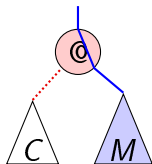
Handling coercions

There are several ways of handling coercions.

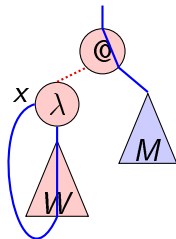
- ▶ As erasable contexts: coercions appear around terms.
- ▶ As functions: coercions apply on terms.
- ▶ As functions and as erasable contexts.



Erasable context only



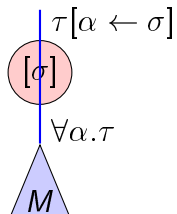
Functions only



Both

Description of F_η

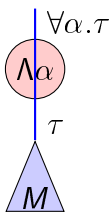
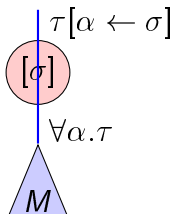
In F_η , there are three ways to change a type:



instantiation

Description of F_η

In F_η , there are three ways to change a type:

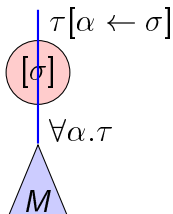


instantiation

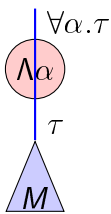
generalization

Description of F_η

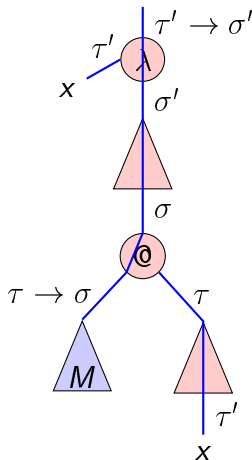
In F_η , there are three ways to change a type:



instantiation



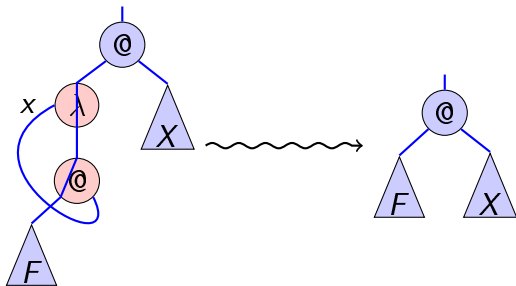
generalization



η -expansion

Intuition of the ETA rule

η -expansions do not change the semantics of a term. But they allow to retype a posteriori on both side of the arrow, as we saw.



So Mitchell came with the idea of *retyping functions*, which are functions η -equivalent to the identity function. These functions allow to deeply instantiate a type in a contra-variant way.

F_η 's proof term variant

Instead of having coercions be lambda terms, we can use proof terms of a type containment with the following rules as Mitchell showed:

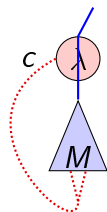
$$\frac{\text{DIST}}{\overrightarrow{\forall\alpha.(\tau \rightarrow \sigma)} \subseteq (\overrightarrow{\forall\alpha.\tau}) \rightarrow \overrightarrow{\forall\alpha.\sigma}} \qquad \frac{\text{ARROW} \quad \sigma' \subseteq \sigma \quad \tau \subseteq \tau'}{\sigma \rightarrow \tau \subseteq \sigma' \rightarrow \tau'}$$

$$\frac{\text{SUB} \quad \overrightarrow{\beta} \notin \text{ftv}(\overrightarrow{\forall\alpha.\sigma})}{\overrightarrow{\forall\alpha.\sigma} \subseteq \overrightarrow{\forall\beta.\sigma[\alpha \leftarrow \tau]}} \qquad \frac{\text{TRANS} \quad \rho \subseteq \sigma \quad \sigma \subseteq \tau}{\rho \subseteq \tau} \qquad \frac{\text{CONGRUENCE} \quad \sigma \subseteq \tau}{\forall\alpha.\sigma \subseteq \forall\alpha.\tau}$$

However this version is not practical to program with.

Description of xML^F

xML^F allows to abstract over coercions:



For example, a term typed with $\forall(\alpha \geq \sigma_{id})\alpha \rightarrow \alpha$ can be instantiated with a coercion from σ_{id} to α . This allows to do deep covariant instantiation.

There is a lambda-term version of xML^F (F_c [Manzonetto and Tranquilli '10]).

Bisimulation

We saw that coercions allow to change the type of a term in several ways, but we also want coercions to be erasable without loosing the semantics. This means that we have two reduction steps in the source language:

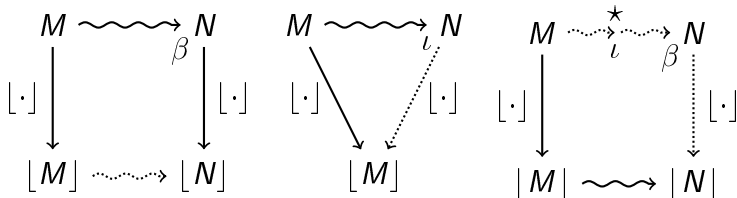
- ▶ the computation step β and
- ▶ the coercion related step ι .

Bisimulation

We saw that coercions allow to change the type of a term in several ways, but we also want coercions to be erasable without losing the semantics. This means that we have two reduction steps in the source language:

- ▶ the computation step β and
- ▶ the coercion related step ι .

Having a bisimulation up to ι between the source language and its erasure means to have the following properties:



Goals

Is it possible to unify these different possibilities of changing a type a posteriori? Said otherwise, is it possible to add coercion abstraction to F_η , and still preserve the coercion-erasure semantics?

This is challenging for several (and sometimes unexpected) reasons:

- ▶ The drop function is surprisingly not obvious.
- ▶ Labeling transitions with β or ι is also problematic.
- ▶ And bisimulation needs some restrictions.

Ideas

We add on top of retyping functions the idea of erasable context.

During $F_{\lambda\eta}$'s design, we keep in mind the following points:

- ▶ Bisimulation (because this is the hard part which is guiding every choice):
 - ▶ β -reduction drops on β -reduction,
 - ▶ ι -reduction drops on equality, and
 - ▶ β -reduction comes from ι -reductions followed by a β -reduction.
- ▶ Strong normalization: stay as close to F as possible.
- ▶ Soundness: subject reduction and progress.

In the following, the erasure of a term can be directly read from its blue parts.

Syntax and notations

We only need colors on the edges. But we add colors on the nodes for readability. The node colors can be deduced locally from the edge colors.

Syntax and notations

We only need colors on the edges. But we add colors on the nodes for readability. The node colors can be deduced locally from the edge colors.

graphical syntax



Syntax and notations

We only need colors on the edges. But we add colors on the nodes for readability. The node colors can be deduced locally from the edge colors.

graphical syntax




textual syntax

$\lambda^{\square} \underline{x}. \underline{M}$

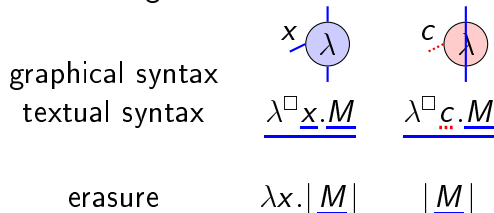
Syntax and notations

We only need colors on the edges. But we add colors on the nodes for readability. The node colors can be deduced locally from the edge colors.

graphical syntax	
textual syntax	$\lambda^{\square} \underline{x}. \underline{M}$
erasure	$\lambda x. [\underline{M}]$

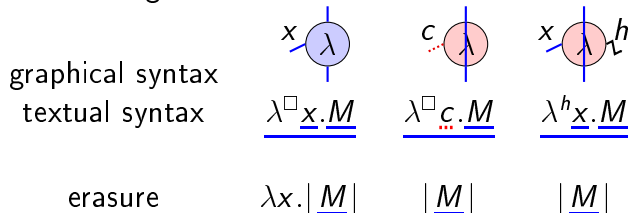
Syntax and notations

We only need colors on the edges. But we add colors on the nodes for readability. The node colors can be deduced locally from the edge colors.



Syntax and notations

We only need colors on the edges. But we add colors on the nodes for readability. The node colors can be deduced locally from the edge colors.



Syntax and notations

We only need colors on the edges. But we add colors on the nodes for readability. The node colors can be deduced locally from the edge colors.

graphical syntax				
textual syntax	$\lambda^{\square} \underline{x}. \underline{M}$	$\lambda^{\square} \underline{c}. \underline{M}$	$\lambda^h \underline{x}. \underline{M}$	$\lambda^{\square} \underline{z}. \underline{M}$
erasure	$\lambda x. \underline{[M]}$	$\underline{[M]}$	$\underline{[M]}$	undef

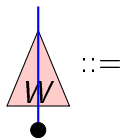
Syntax and notations

We only need colors on the edges. But we add colors on the nodes for readability. The node colors can be deduced locally from the edge colors.

graphical syntax				
textual syntax	$\lambda^{\square} \underline{x}. \underline{M}$	$\lambda^{\square} \underline{c}. \underline{M}$	$\lambda^h \underline{x}. \underline{M}$	$\lambda^{\square} \underline{z}. \underline{M}$
erasure	$\lambda x. \underline{[M]}$	$\underline{[M]}$	$\underline{[M]}$	undef
graphical syntax				
textual syntax	$\underline{M} @^{\square} \underline{N}$	$\underline{M} @^{\square} \underline{N}$	$\underline{M} @^h \underline{N}$	$\underline{M} @^{\square} \underline{N}$
erasure	$\underline{[M]} \underline{[N]}$	$\underline{[M]}$	$\underline{[M]}$	$\underline{[N]}$

Understanding erasable contexts

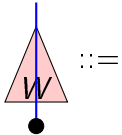
We define valid erasable contexts by induction with following grammar rule.



identity				

Understanding erasable contexts

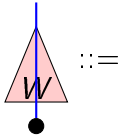
We define valid erasable contexts by induction with following grammar rule.



identity	coercion contexts already in F_c			

Understanding erasable contexts

We define valid erasable contexts by induction with following grammar rule.



identity	coercion contexts already in F_c			η -expansion

Sketching the type system

We want an erasable context judgment $\Gamma; \bullet : \underline{\tau} \vdash \triangle_W : \underline{\sigma}$.

But because we don't want the following deep typing rule

$$\frac{\Gamma; \underline{\cdot} : \underline{\tau} \vdash W_3 [\underline{\cdot}] : \underline{\tau}_3 \quad \Gamma; \underline{\cdot} : \underline{\tau}_2 \vdash W_1 [\underline{\cdot}] : \underline{\sigma}}{\Gamma; \underline{\cdot} : \underline{\tau}_1 \vdash W_2 [\underline{\cdot}] : \underline{\tau}_3 \rightarrow \underline{\tau}_2 \quad \Gamma; \underline{\cdot} : \underline{\tau}_2 \vdash W_1 [\underline{\cdot}] : \underline{\sigma}}$$

$$\frac{\Gamma; \underline{\cdot} : \underline{\tau} \vdash W_3 [\underline{\cdot}] : \underline{\tau}_3 \quad \Gamma; \underline{\cdot} : \underline{\tau}_2 \vdash W_1 [\underline{\cdot}] : \underline{\sigma}}{\Gamma; \underline{\cdot} : \underline{\tau}_1 \vdash \lambda^h x. W_1 \left[\underline{W_2 [\underline{\cdot}]} @^h W_3 [\underline{x}] \right] : \underline{\tau} \rightarrow \underline{\sigma}}$$

we need a way to type $W_1 [\cdot]$ remembering we are after a λ^h so that the context is filled with a $@^h$. So we introduce an environment describing the stack of λ^h we went through and didn't see their $@^h$ yet.

Sketching the type system

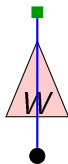
$$\Sigma; \Gamma; Z; \Delta \vdash M : \tau$$

Σ contains the free type variables. Γ contains the free term variables.

Sketching the type system

$$\Sigma; \Gamma; Z; \Delta \vdash M : \tau$$

The $Z \times \Delta$ combination reads like this:



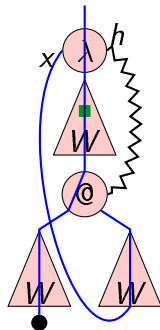
$(\underline{z} : \underline{\tau}) \times \emptyset$: We type a valid erasable context
(i.e. $W[\cdot]$) where the hole is occupied by \underline{z} .

Sketching the type system

$$\Sigma; \Gamma; Z; \Delta \vdash M : \tau$$

The $Z \times \Delta$ combination reads like this:

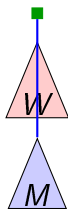
$(\underline{z} : \underline{\tau}) \times (\Delta, (h : \underline{x} : \underline{\sigma}))$: We type a valid erasable context where the hole is occupied by $\underline{W}_\Delta [\underline{z}] @^h \underline{W} [\underline{x}]$



Sketching the type system

$$\Sigma; \Gamma; Z; \Delta \vdash M : \tau$$

The $Z \times \Delta$ combination reads like this:



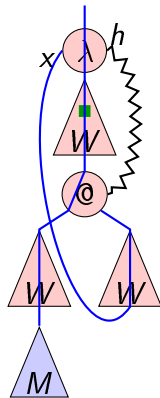
$\emptyset \times \emptyset$: We type a normal term that we can see as a valid erasable context of a normal term.

Sketching the type system

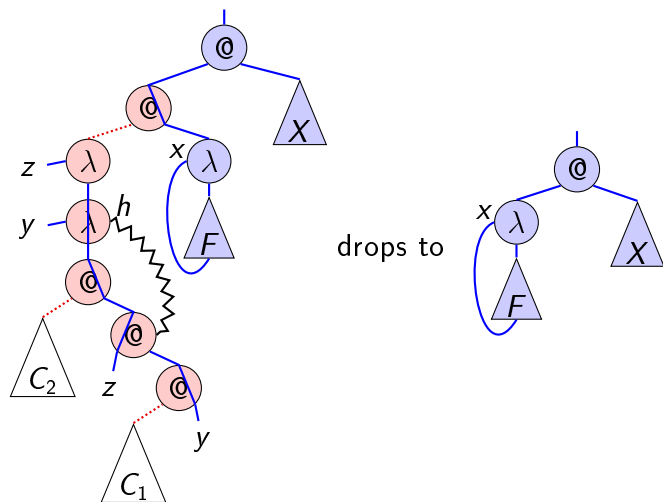
$$\Sigma; \Gamma; Z; \Delta \vdash M : \tau$$

The $Z \times \Delta$ combination reads like this:

$\emptyset \times (\Delta, (h : \underline{x} : \underline{\tau}))$: We type a valid erasable context of $\underline{M@^h W [\underline{x}]}$ where \underline{M} is a Δ -valid erasable context of a normal term.

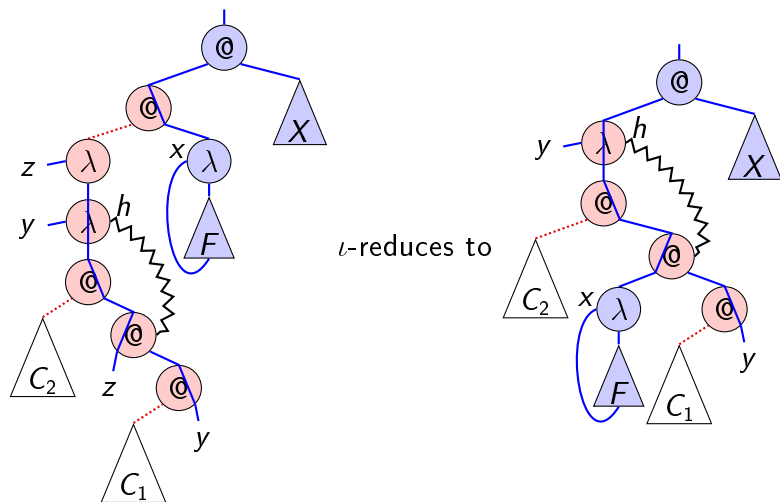


Sketching the reduction rules



We want to coerce F using C_1 on its argument (left side of the arrow) and C_2 on its result (right side of the arrow).

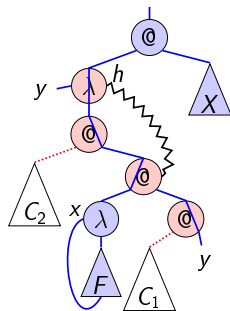
Sketching the reduction rules



We reduce the only redex. It's not a β -reduction, so we have the same erasure.

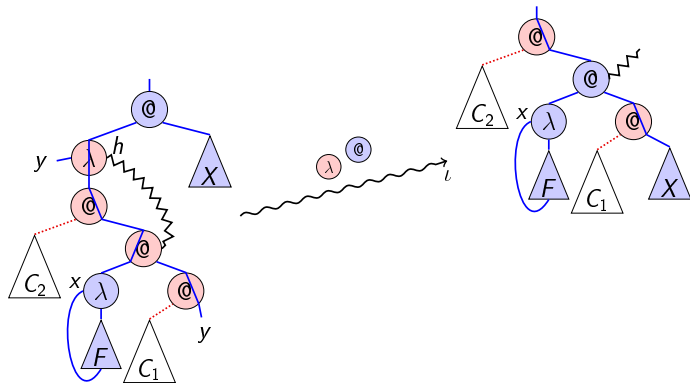
Sketching the reduction rules

We have two redexes involving the same flash.



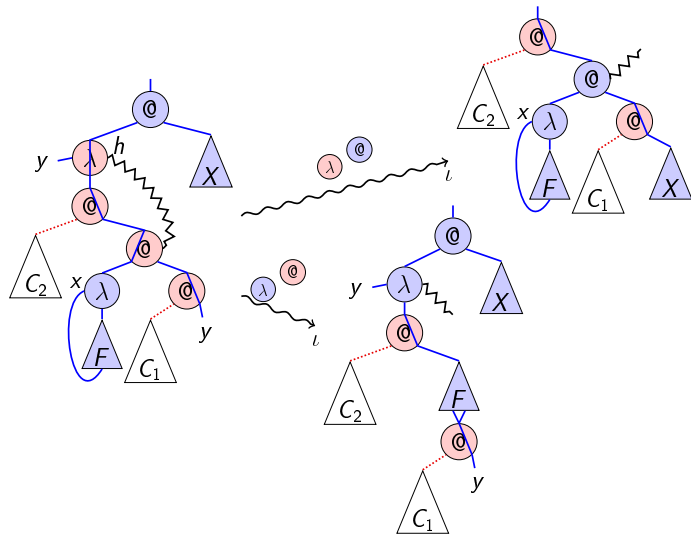
Sketching the reduction rules

We have two redexes involving the same flash.



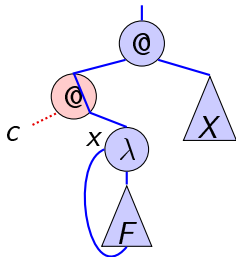
Sketching the reduction rules

We have two redexes involving the same flash.



Restriction

The bisimulation doesn't work so easily because we can have the following situation where a coercion variable blocks a β -redex:



To avoid this, we use a trick similar to the one in F_c , which mimics xML^F and only allow abstraction over coercions that are parametric on their return type. This example would then be ill-typed.

Restriction

We have the following typing rule:

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

which does two things:

- ▶ it abstracts over the coercion \underline{x} and
- ▶ it generalizes the return type of the coercion.

Restriction

We have the following typing rule:

$$\text{CABS} \frac{\Sigma, \underline{\alpha}; \Gamma, (x : \underline{\tau} \rightarrow \underline{\alpha}); Z; \Delta \vdash \underline{M} : \underline{\sigma}}{\Sigma; \Gamma; Z; \Delta \vdash \lambda^{\square} x. \underline{M} : \forall \alpha. (\underline{\tau} \rightarrow \underline{\alpha}) \rightarrow \underline{\sigma}}$$

which does two things:

- ▶ it abstracts over the coercion \underline{x} and
- ▶ it generalizes the return type of the coercion.

This last part was absent in F_c which breaks subject reduction.

$$\text{FCCABS} \frac{\Gamma, (x : \tau \multimap \alpha); L \vdash_{\text{tl}} a : \sigma}{\Gamma; L \vdash_{\text{tl}} \underline{\lambda} x. a : (\tau \multimap \alpha) \rightarrow \sigma}$$

Restriction

$$\begin{aligned}\Gamma &= (b : \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha), (f : \beta \rightarrow \beta) \\ a &= (\underline{\lambda}c_1. \underline{\lambda}c_2. b (c_1 \triangleright f) (c_2 \triangleright f)) \triangleleft (\underline{\lambda}z. z) \\ a' &= \underline{\lambda}c_2. b ((\underline{\lambda}z. z) \triangleright f) (c_2 \triangleright f) \\ b &= \underline{\lambda}c_2. b f (c_2 \triangleright f) \\ \tau &= ((\beta \rightarrow \beta) \multimap (\beta \rightarrow \beta)) \rightarrow (\beta \rightarrow \beta)\end{aligned}$$

We have $\Gamma; \vdash_{\mathfrak{t}} a : \tau$ and $a \rightarrow_c a' \rightarrow_c b$, but we don't have $\Gamma; \vdash_{\mathfrak{t}} b : \tau$ because the return type of the coercion is an arrow instead of a variable.

Restriction

$$\begin{aligned}\Gamma &= (b : \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha), (f : \beta \rightarrow \beta) \\ a &= (\underline{\lambda}c_1. \underline{\lambda}c_2. b (c_1 \triangleright f) (c_2 \triangleright f)) \triangleleft (\underline{\lambda}z. z) \\ a' &= \underline{\lambda}c_2. b ((\underline{\lambda}z. z) \triangleright f) (c_2 \triangleright f) \\ b &= \underline{\lambda}c_2. b f (c_2 \triangleright f) \\ \tau &= ((\beta \rightarrow \beta) \multimap (\beta \rightarrow \beta)) \rightarrow (\beta \rightarrow \beta)\end{aligned}$$

We have $\Gamma; \vdash_t a : \tau$ and $a \rightarrow_c a' \rightarrow_c b$, but we don't have $\Gamma; \vdash_t b : \tau$ because the return type of the coercion is an arrow instead of a variable.

Enforcing α 's generalization just after the abstraction like it is done in CABS (and xML^F) rejects such counter-examples.

Conclusion

- ▶ We have shown how to extend F_η with coercion abstraction.
- ▶ The meta-theory is not so deep, as it essentially works like System F.
- ▶ However, the technical details are still involved, due to the well-formedness conditions.
- ▶ The source of the difficulties were quite unexpected.

Extensions

- ▶ Hopefully, the presentation (syntax) could be simplified breadthwise.
- ▶ The restriction on coercion abstraction might be weakened or removed.
- ▶ We could also add more constructors like functions between coercions $\lambda^{\square} c.M$, functions returning coercions $\lambda^{\square} x.M$, or coercions between coercions $\lambda^{\square} z.M$. For example $\lambda^{\square} c_1.\lambda^{\square} c_2.\lambda^{\square} f.\lambda^h x.c_2 @^{\square} (f @^h (c_1 @^{\square} x))$. Such terms were considered in our first approach.
- ▶ We could extend retyping functions by adding β -equivalence. This way we could program in coercions, and for example use the polymorphic `List.map` function to build a coercion $\alpha \text{ list} \rightarrow \beta \text{ list}$ from a coercion $\alpha \rightarrow \beta$.