Erasable coercions: a unified approach to type systems

Julien Cretin

January 30, 2014

Background: machine language

A machine

- executes programs
- written in machine language

Background: machine language

A machine

- executes programs
- written in machine language

Problem

- programs may crash, and
- programs in machine language are hard to write.

Background: high-level language

In practice, programs are

- written in high-level languages (C, Java, OCaml)
- compiled to low-level ones (x86, ARM, bytecode)

Background: high-level language

In practice, programs are

- written in high-level languages (C, Java, OCaml)
- compiled to low-level ones (x86, ARM, bytecode)

Thus, programs are

- easier to write
- more portable

Background: high-level language

In practice, programs are

- written in high-level languages (C, Java, OCaml)
- compiled to low-level ones (x86, ARM, bytecode)

Thus, programs are

- easier to write
- more portable

Problem

For most high-level languages, programs may still crash.

Background: type system

Therefore, languages come with a type system.

A type system

- defines program type annotations
- ensures that well-typed programs do not crash

Background: type system

Therefore, languages come with a type system.

A type system

- defines program type annotations
- ensures that well-typed programs do not crash

The compiler of a typed language

- takes as input a program with type annotations,
- checks that the program is well-typed,
- erases type annotations, and
- compiles the program.

Compiled well-typed programs do not crash (if the compiler is correct).

In my PhD

I consider one particular language:

Lambda Calculus

- a minimal model of computation
- at the root of functional programming languages (OCaml, Haskell, ...)

Still, our approach is not restricted to the Lambda Calculus.

Type systems are a maze

There are many type systems just for the Lambda Calculus:

- ML
- System F
- System F-sub
- System F-omega
- System F-eta
- MLF
- ... (the list grows as we speak)

Type systems are a maze

There are many type systems just for the Lambda Calculus:

- ML
- System F
- System F-sub
- System F-omega
- System F-eta
- MLF
- ... (the list grows as we speak)

Why does one language need so many type systems?

Type systems are compromises

Comparison criteria

expressiveness:	what is the set of well-typed terms?
inference:	how many annotations may be omitted?
simplicity:	are annotations simple to write and understand?

Type systems are compromises

Comparison criteria

expressiveness:	what is the set of well-typed terms?
inference:	how many annotations may be omitted?
simplicity:	are annotations simple to write and understand?

Inference and simplicity

- mainly matter for surface type systems
- do not matter for kernel type systems

Type systems are compromises

Comparison criteria

expressiveness:	what is the set of well-typed terms?
inference:	how many annotations may be omitted?
simplicity:	are annotations simple to write and understand?

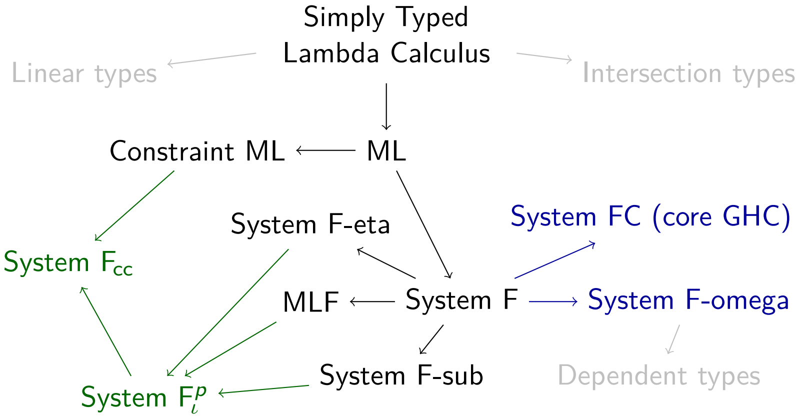
Inference and simplicity

- mainly matter for surface type systems
- do not matter for kernel type systems

We study kernel type systems.

- A kernel type system may have different surface type systems.
- Translating from surface type systems to kernel type systems is easy.
- Kernel type systems are better suited for meta-theory.

Kernel type systems



Type system features

polymorphism, η-expansion, subtyping, and coercions.

Outline

Prelude Lambda Calculus

- definition
- example

Outline

Prelude Lambda Calculus

- definition
- example

Part I Syntactical approach (System F_ι^p)

- identify a set of atomic features
- present them as composable coercions
- define a unified framework of coercions

Outline

Prelude Lambda Calculus

- definition
- example

Part I Syntactical approach (System F_{ι}^{p})

- identify a set of atomic features
- present them as composable coercions
- define a unified framework of coercions

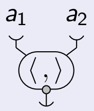
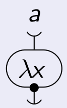
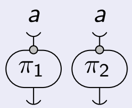
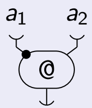
Part II Semantical approach (System F_{cc})

- general coercion abstraction
- first-class coercions

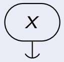
Lambda Calculus: terms

We only study pairs and functions, but the usual constructs may be added.

Terms

(active edge)	Product (gray)	Arrow (black)
Constructors (output)	pairs 	abstractions 
Destructors (input)	projections 	applications 

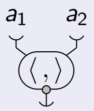
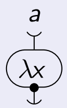
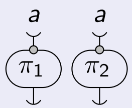
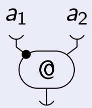
variables



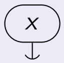
Lambda Calculus: terms

We only study pairs and functions, but the usual constructs may be added.

Terms

(active edge)	Product (gray)	Arrow (black)
Constructors (output)	pairs 	abstractions 
Destructors (input)	projections 	applications 

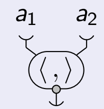
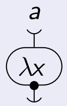
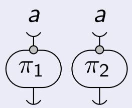
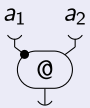
variables



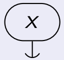
Lambda Calculus: terms

We only study pairs and functions, but the usual constructs may be added.

Terms

(active edge)	Product (gray)	Arrow (black)
Constructors (output)	pairs 	abstractions 
Destructors (input)	projections 	applications 

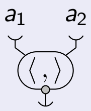
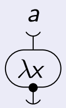
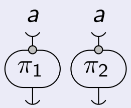
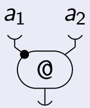
variables



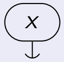
Lambda Calculus: terms

We only study pairs and functions, but the usual constructs may be added.

Terms

(active edge)	Product (gray)	Arrow (black)
Constructors (output)	pairs 	abstractions 
Destructors (input)	projections 	applications 

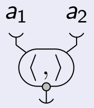
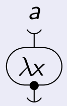
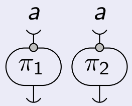
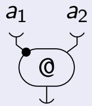
variables



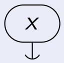
Lambda Calculus: terms

We only study pairs and functions, but the usual constructs may be added.

Terms

(active edge)	Product (gray)	Arrow (black)
Constructors (output)	pairs 	abstractions 
Destructors (input)	projections 	applications 

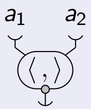
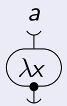
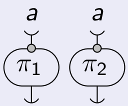
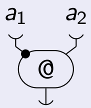
variables



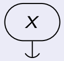
Lambda Calculus: terms

We only study pairs and functions, but the usual constructs may be added.

Terms

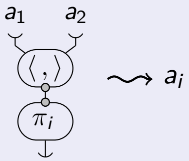
(active edge)	Product (gray)	Arrow (black)
Constructors (output)	pairs 	abstractions 
Destructors (input)	projections 	applications 

variables



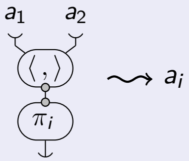
Lambda Calculus: reduction

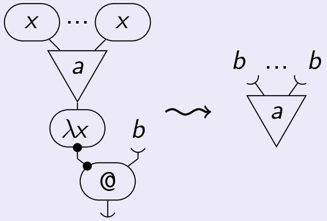
Reduction (when two active edges of the same sort meet)



Lambda Calculus: reduction

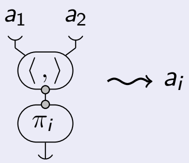
Reduction (when two active edges of the same sort meet)

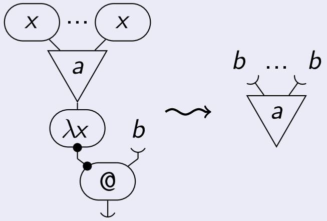




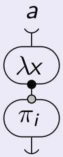
Lambda Calculus: reduction

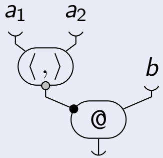
Reduction (when two active edges of the same sort meet)



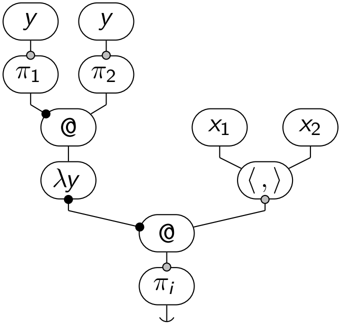


Errors (when two active edges of different sorts meet)

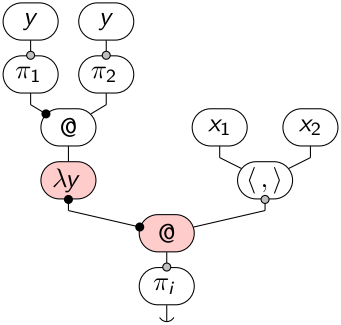




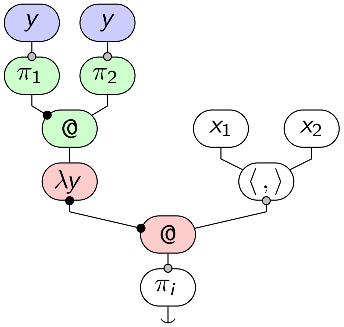
Lambda Calculus: example



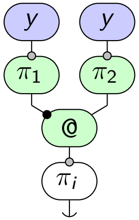
Lambda Calculus: example



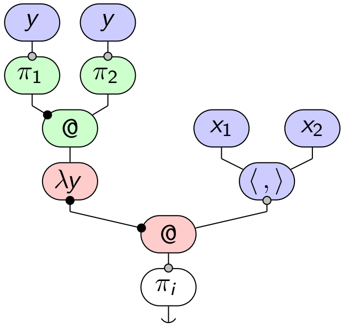
Lambda Calculus: example



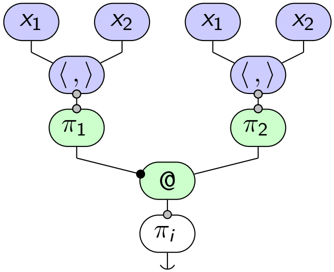
$\rightsquigarrow$



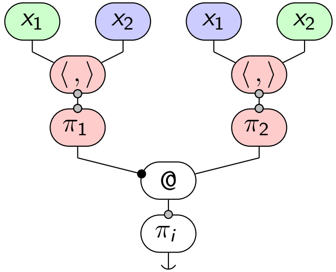
Lambda Calculus: example



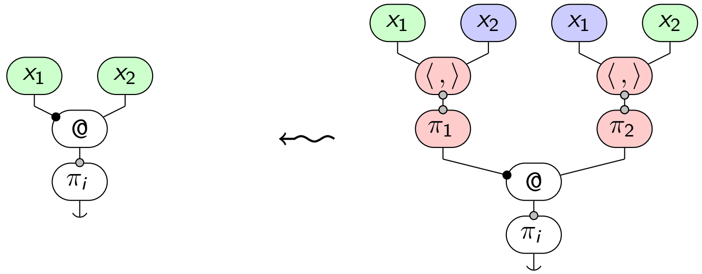
$\leadsto$



Lambda Calculus: example



Lambda Calculus: example



Strong reduction

Although all programming languages execution is in weak reduction (reduction does not proceed under abstractions),
we should study meta-theoretical properties for strong reduction.

Strong reduction

Although all programming languages execution is in weak reduction
(reduction does not proceed under abstractions),
we should study meta-theoretical properties for strong reduction.

We may reduce

- anywhere (in particular under abstractions)
- in any order

Strong reduction

Although all programming languages execution is in weak reduction (reduction does not proceed under abstractions),
we should study meta-theoretical properties for strong reduction.

We may reduce

- anywhere (in particular under abstractions)
- in any order

Studying strong reduction is much harder, but it

- gives a better understanding of the language and type system
- looks at potentially hidden errors as usual type systems do ($\lambda x \, \langle x, x \rangle \; x$ is an error in strong reduction, but not in weak reduction)
- justifies a certain class of optimizations
- describes all reduction strategies (in particular weak ones)

Part I: Syntactical approach

We now can define our unified *framework of coercions*

- identifying a set of atomic features
- presenting them as composable coercions

This approach is syntactical.

Framework idea: we distinguish two sorts of typing rules

Computational rules

$$\frac{(\Gamma, \Sigma_i \vdash a_i : \tau_i)^{i \in 1..n} \qquad J_1 \ldots J_m}{\Gamma \vdash node(a_1, \ldots a_n) : \rho}$$

Erasable rules

$$\frac{\Gamma, \Sigma \vdash a : \tau \qquad J_1 \ldots J_m}{\Gamma \vdash a : \rho}$$

Framework idea: we distinguish two sorts of typing rules

Computational rules

$$\frac{(\Gamma, \Sigma_i \vdash a_i : \tau_i)^{i \in 1..n} \qquad J_1 \ldots J_m}{\Gamma \vdash node(a_1, \ldots a_n) : \rho}$$

Erasable rules

$$\frac{\Gamma, \Sigma \vdash a : \tau \qquad J_1 \ldots J_m}{\Gamma \vdash a : \rho}$$

We factor all erasable rules into a unique erasable rule:

Unique erasable rule

TermCoer

$$\frac{\Gamma, \Sigma \vdash a : \tau \qquad \Gamma \vdash (\Sigma \vdash \tau) \triangleright \rho}{\Gamma \vdash a : \rho}$$

Coercion rules

$$\frac{J_1 \ldots J_m}{\Gamma \vdash (\Sigma \vdash \tau) \triangleright \rho}$$

Framework idea: we distinguish two sorts of typing rules

Computational rules

$$\frac{(\Gamma, \Sigma_i \vdash a_i : \tau_i)^{i \in 1..n} \qquad J_1 \ldots J_m}{\Gamma \vdash \mathit{node}(a_1, \ldots a_n) : \rho}$$

Erasable rules

$$\frac{\Gamma, \Sigma \vdash a : \tau \qquad J_1 \ldots J_m}{\Gamma \vdash a : \rho}$$

We factor all erasable rules into a unique erasable rule:

Unique erasable rule

TermCoer

$$\frac{\Gamma, \Sigma \vdash a : \tau \qquad \Gamma \vdash (\Sigma \vdash \tau) \triangleright \rho}{\Gamma \vdash a : \rho}$$

Coercion rules

$$\frac{J_1 \ldots J_m}{\Gamma \vdash (\Sigma \vdash \tau) \triangleright \rho}$$

- The unique erasable rule looks like the usual term subtyping rule.
- Besides changing the type, coercions may also extend the environment.

Framework gain: composable features

It looks like we only played with the syntax of typing rules.

Framework gain: composable features

It looks like we only played with the syntax of typing rules.

However, this change makes a clear distinction between terms and type annotations: terms are totally absent from coercion rules.

$$\frac{J_1 \dots J_m}{\Gamma \vdash (\Sigma \vdash \tau) \triangleright \rho}$$

Framework gain: composable features

It looks like we only played with the syntax of typing rules.

However, this change makes a clear distinction between terms and type annotations: terms are totally absent from coercion rules.

$$\frac{J_1 \ldots J_m}{\Gamma \vdash (\Sigma \vdash \tau) \triangleright \rho}$$

More essentially,

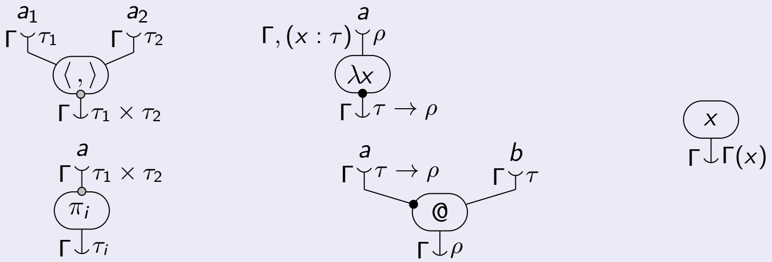
- rule TermCoer enforces a unique interface for all erasable features:

$$\Gamma \vdash (\Sigma \vdash \tau) \triangleright \rho$$

- this change is a preliminary to have composable features.
- this enables to decompose existing features into atomic parts.

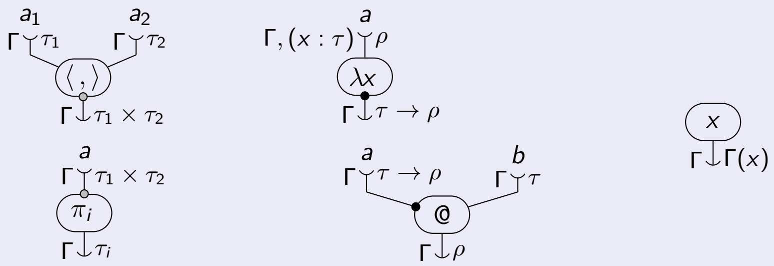
Framework: all term typing rules

STLC (canonical, minimal set of typing rules)

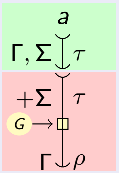

a_1
$\Gamma \curlyvee \tau_1$
a_2
$\Gamma \curlyvee \tau_2$
$\langle\, ,\rangle$
$\Gamma \downarrow \tau_1 \times \tau_2$
a
$\Gamma, (x : \tau) \curlyvee \rho$
λx
$\Gamma \downarrow \tau \to \rho$
x
$\Gamma \downarrow \Gamma(x)$
a
$\Gamma \curlyvee \tau_1 \times \tau_2$
π_i
$\Gamma \downarrow \tau_i$
a
$\Gamma \curlyvee \tau \to \rho$
b
$\Gamma \curlyvee \tau$
@
$\Gamma \downarrow \rho$

Framework: all term typing rules

STLC (canonical, minimal set of typing rules)



Term coercion rule



$$\left(\begin{array}{c} \textsc{TermCoer} \\ \dfrac{\Gamma, \Sigma \vdash a : \tau \qquad G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \triangleright \rho}{\Gamma \vdash a : \rho} \end{array} \right)$$

Feature: polymorphism

We write $\tau \triangleright \rho$ for $(\varnothing \vdash \tau) \triangleright \rho$. This is actually how subtyping is encoded.

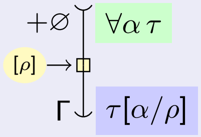
Feature: polymorphism

We write $\tau \triangleright \rho$ for $(\varnothing \vdash \tau) \triangleright \rho$. This is actually how subtyping is encoded.

Coercion rules

OldInst

$$\frac{\Gamma \vdash a : \forall\alpha\,\tau}{\Gamma \vdash a : \tau[\alpha/\rho]}$$



Inst

$$[\rho] \Rightarrow \Gamma \vdash \forall\alpha\,\tau \triangleright \tau[\alpha/\rho]$$

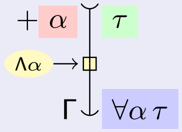
Feature: polymorphism

We write $\tau \triangleright \rho$ for $(\varnothing \vdash \tau) \triangleright \rho$. This is actually how subtyping is encoded.

Coercion rules

OldGen

$$\frac{\Gamma, \alpha \vdash a : \tau}{\Gamma \vdash a : \forall\alpha\,\tau}$$

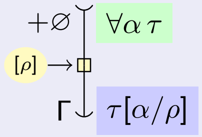


Gen

$\Lambda\alpha \Rightarrow \Gamma \vdash (\alpha \vdash \tau) \triangleright \forall\alpha\,\tau$

OldInst

$$\frac{\Gamma \vdash a : \forall\alpha\,\tau}{\Gamma \vdash a : \tau[\alpha/\rho]}$$



Inst

$[\rho] \Rightarrow \Gamma \vdash \forall\alpha\,\tau \triangleright \tau[\alpha/\rho]$

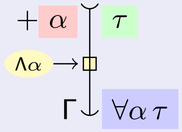
Feature: polymorphism

We write $\tau \triangleright \rho$ for $(\varnothing \vdash \tau) \triangleright \rho$. This is actually how subtyping is encoded.

Coercion rules

OldGen

$$\frac{\Gamma, \alpha \vdash a : \tau}{\Gamma \vdash a : \forall \alpha\, \tau}$$

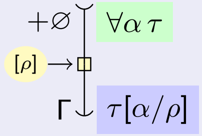


Gen

$\Lambda\alpha \;\Rightarrow \Gamma \vdash (\alpha \vdash \tau) \triangleright \forall\alpha\,\tau$

OldInst

$$\frac{\Gamma \vdash a : \forall \alpha\, \tau}{\Gamma \vdash a : \tau[\alpha/\rho]}$$



Inst

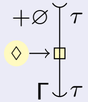
$[\rho] \;\Rightarrow \Gamma \vdash \forall\alpha\,\tau \triangleright \tau[\alpha/\rho]$

STLC + polymorphism = System F

Extending the framework

We internalize the reflexivity and transitivity of coercions.

Coercion rules



Refl

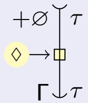
$\lozenge \Rightarrow \Gamma \vdash \tau \rhd \tau$

These rules are now part of the framework.

Extending the framework

We internalize the reflexivity and transitivity of coercions.

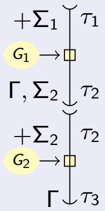
Coercion rules



Refl

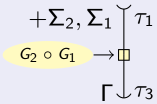
$$\Diamond \Rightarrow \Gamma \vdash \tau \triangleright \tau$$

if



premises

then



conclusion

Trans

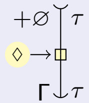
$$\frac{G_2 \Rightarrow \Gamma \vdash (\Sigma_2 \vdash \tau_2) \triangleright \tau_3 \qquad G_1 \Rightarrow \Gamma, \Sigma_2 \vdash (\Sigma_1 \vdash \tau_1) \triangleright \tau_2}{G_2 \circ G_1 \Rightarrow \Gamma \vdash (\Sigma_2, \Sigma_1 \vdash \tau_1) \triangleright \tau_3}$$

These rules are now part of the framework.

Extending the framework

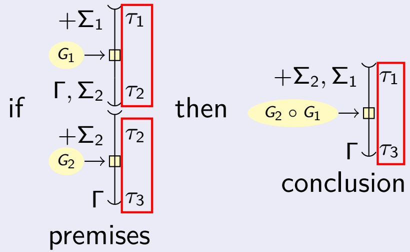
We internalize the reflexivity and transitivity of coercions.

Coercion rules



Refl

$$\Diamond \Rightarrow \Gamma \vdash \tau \triangleright \tau$$



Trans

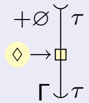
$$\frac{\begin{array}{c} G_2 \Rightarrow \Gamma \vdash (\Sigma_2 \vdash \tau_2) \triangleright \tau_3 \\ G_1 \Rightarrow \Gamma, \Sigma_2 \vdash (\Sigma_1 \vdash \tau_1) \triangleright \tau_2 \end{array}}{G_2 \circ G_1 \Rightarrow \Gamma \vdash (\Sigma_2, \Sigma_1 \vdash \tau_1) \triangleright \tau_3}$$

These rules are now part of the framework.

Extending the framework

We internalize the reflexivity and transitivity of coercions.

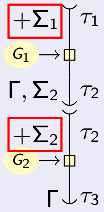
Coercion rules



Refl

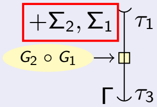
$$\lozenge \Rightarrow \Gamma \vdash \tau \rhd \tau$$

if



premises

then



conclusion

Trans

$$\frac{\begin{array}{c} G_2 \Rightarrow \Gamma \vdash (\Sigma_2 \vdash \tau_2) \rhd \tau_3 \\ G_1 \Rightarrow \Gamma, \Sigma_2 \vdash (\Sigma_1 \vdash \tau_1) \rhd \tau_2 \end{array}}{G_2 \circ G_1 \Rightarrow \Gamma \vdash (\Sigma_2, \Sigma_1 \vdash \tau_1) \rhd \tau_3}$$

These rules are now part of the framework.

Feature: eta-expansion (rules)

Eta-expansion is at the root of subtyping.

Coercion rules

EtaProd

$$\frac{\Gamma \vdash (\Sigma \vdash \tau_1) \triangleright \rho_1 \qquad \Gamma \vdash (\Sigma \vdash \tau_2) \triangleright \rho_2}{\Gamma \vdash (\Sigma \vdash \tau_1 \times \tau_2) \triangleright \rho_1 \times \rho_2}$$

EtaArr

$$\frac{\Gamma, \Sigma \vdash \rho_1 \triangleright \tau_1 \qquad \Gamma \vdash (\Sigma \vdash \tau_2) \triangleright \rho_2}{\Gamma \vdash (\Sigma \vdash \tau_1 \to \tau_2) \triangleright \rho_1 \to \rho_2}$$

These coercion rules extend the subtyping congruence rules of computational types to our notion of coercions.

Note: You need one such rule for each of your computational types.

Feature: eta-expansion (rules)

Eta-expansion is at the root of subtyping.

Coercion rules

EtaProd

$$\frac{G_1 \Rightarrow \Gamma \vdash (\Sigma \vdash \tau_1) \triangleright \rho_1 \qquad G_2 \Rightarrow \Gamma \vdash (\Sigma \vdash \tau_2) \triangleright \rho_2}{G_1 \times G_2 \Rightarrow \Gamma \vdash (\Sigma \vdash \tau_1 \times \tau_2) \triangleright \rho_1 \times \rho_2}$$

EtaArr

$$\frac{G_1 \Rightarrow \Gamma, \Sigma \vdash \rho_1 \triangleright \tau_1 \qquad G_2 \Rightarrow \Gamma \vdash (\Sigma \vdash \tau_2) \triangleright \rho_2}{G_1 \to G_2 \Rightarrow \Gamma \vdash (\Sigma \vdash \tau_1 \to \tau_2) \triangleright \rho_1 \to \rho_2}$$

These coercion rules extend the subtyping congruence rules of computational types to our notion of coercions.

Note: You need one such rule for each of your computational types.

Feature: eta-expansion (consequences)

STLC $+$ polymorphism $+$ eta-expansion $\equiv$ System F_η (Mitchell 1988)

Feature: eta-expansion (consequences)

STLC + polymorphism + eta-expansion $\equiv$ System F_η (Mitchell 1988)

Our presentation is improved:

(thanks to the use of coercions instead of just subtyping)

- polymorphic type congruence rule is derivable:

$$+\alpha\;\dfrac{\dfrac{\dfrac{\forall\alpha\,\tau}{\tau}\,[\alpha]}{\rho}\,G}{\forall\alpha\,\rho}\,\Lambda\alpha \qquad\qquad \Lambda\alpha \circ G \circ [\alpha]$$

- distributivity rules are derivable:

$$([\alpha] \to \Lambda\alpha) \circ [\alpha] \qquad\qquad (\Lambda\alpha \times \Lambda\alpha) \circ [\alpha]$$

Feature: lower bounded polymorphism

Lower bounded polymorphism is a form of coercion abstraction.

Coercion rules

LowerInst

$$\frac{\Gamma \vdash (\sigma \rhd \alpha)[\alpha/\rho]}{\Gamma \vdash \forall(\alpha \lhd \sigma)\,\tau \;\; \rhd \;\; \tau[\alpha/\rho]}$$

LowerGen

$$\Gamma \vdash (\alpha \lhd \sigma \vdash \tau) \rhd \forall(\alpha \lhd \sigma)\,\tau$$

Feature: lower bounded polymorphism

Lower bounded polymorphism is a form of coercion abstraction.

Coercion rules

LowerInst

$$\frac{\Gamma \vdash (\sigma \triangleright \alpha)[\alpha/\rho]}{\Gamma \vdash \forall(\alpha \triangleleft \sigma)\,\tau \quad \triangleright \quad \tau[\alpha/\rho]}$$

LowerGen

$$\Gamma \vdash (\alpha \triangleleft \sigma \vdash \tau) \triangleright \forall(\alpha \triangleleft \sigma)\,\tau$$

Feature: lower bounded polymorphism

Lower bounded polymorphism is a form of coercion abstraction.

Coercion rules

LowerInst

$$\frac{\Gamma \vdash (\sigma \triangleright \alpha)[\alpha/\rho]}{\Gamma \vdash \forall(\alpha \triangleleft \sigma)\,\tau \quad \triangleright \quad \tau[\alpha/\rho]}$$

LowerGen

$$\Gamma \vdash (\alpha \triangleleft \sigma \vdash \tau) \triangleright \forall(\alpha \triangleleft \sigma)\,\tau$$

STLC + lower bounded polymorphism
$\equiv$ MLF (Le Botlan, Rémy, Yakobowski, 2003-2010)

This permits to prove the strong normalization of MLF.

Feature: upper bounded polymorphism

Upper bounded polymorphism is also a form of coercion abstraction.

Coercion rules

UpperInst

$$\frac{\Gamma \vdash (\alpha \triangleright \sigma)[\alpha/\rho]}{\Gamma \vdash \forall(\alpha \triangleright \sigma)\,\tau \;\; \triangleright \;\; \tau[\alpha/\rho]}$$

UpperGen

$$\Gamma \vdash (\alpha \triangleright \sigma \vdash \tau) \triangleright \forall(\alpha \triangleright \sigma)\,\tau$$

Feature: upper bounded polymorphism

Upper bounded polymorphism is also a form of coercion abstraction.

Coercion rules

UpperInst
$$\frac{\Gamma \vdash (\alpha \triangleright \sigma)[\alpha/\rho]}{\Gamma \vdash \forall(\alpha \triangleright \sigma)\,\tau \quad \triangleright \quad \tau[\alpha/\rho]}$$

UpperGen
$$\Gamma \vdash (\alpha \triangleright \sigma \vdash \tau) \triangleright \forall(\alpha \triangleright \sigma)\,\tau$$

Feature: upper bounded polymorphism

Upper bounded polymorphism is also a form of coercion abstraction.

Coercion rules

UpperInst
$$\frac{\Gamma \vdash (\alpha \triangleright \sigma)[\alpha/\rho]}{\Gamma \vdash \forall(\alpha \triangleright \sigma)\,\tau \quad \triangleright \quad \tau[\alpha/\rho]}$$

UpperGen
$$\Gamma \vdash (\alpha \triangleright \sigma \vdash \tau) \triangleright \forall(\alpha \triangleright \sigma)\,\tau$$

STLC + eta-expansion + upper bounded polymorphism
$\supsetneq$ System $F_{<:}$ (Canning, Cook, Hill, Olthoff, Mitchell, 1989)

Our version is more expressive than all variants of $F_{<:}$ because coercions are composable, *i.e.* distributivity and congruence rules are derivable.

Properties: bisimulation

We can define a reduction relation $M \rightsquigarrow N$ for explicit terms.
We label reduction steps with

ι (iota)	for erasable steps
β (beta)	for computational steps

Properties: bisimulation

We can define a reduction relation $M \rightsquigarrow N$ for explicit terms.
We label reduction steps with

ι (iota)	for erasable steps
β (beta)	for computational steps

Bisimulation (how explicit and implicit reduction relations relate)

explicit:

implict:

$$\begin{tikzcd}
M \arrow[r, rightsquigarrow, "\beta"'] \arrow[d] & N \arrow[d] \\
a \arrow[r, rightsquigarrow, dotted] & b
\end{tikzcd}
\qquad
\begin{tikzcd}
M \arrow[rr, rightsquigarrow, "\iota"'] \arrow[dr] & & N \arrow[dl, dotted] \\
& a = b &
\end{tikzcd}$$

Forward simulation

$$\begin{tikzcd}
M \arrow[r, rightsquigarrow, dotted, "\star", "\iota"'] \arrow[d] & {} \arrow[r, rightsquigarrow, dotted, "\beta"'] & N \arrow[d, dotted] \\
a \arrow[rr, rightsquigarrow] & & b
\end{tikzcd}$$

Backward simulation

We write a (resp. b) for the type erasure of M (resp. N).

Properties: soundness and strong normalization

We prove soundness with the usual scheme: preservation + progress.

preservation:	reduction preserves well-typedness
progress:	errors are not well-typed

Properties: soundness and strong normalization

We prove soundness with the usual scheme: preservation + progress.

preservation:	reduction preserves well-typedness
progress:	errors are not well-typed

Since explicit terms represent typing derivations, the explicit version of these two lemmas is much simpler to prove than the implicit version.

Properties: soundness and strong normalization

We prove soundness with the usual scheme: preservation + progress.

preservation:	reduction preserves well-typedness
progress:	errors are not well-typed

Since explicit terms represent typing derivations, the explicit version of these two lemmas is much simpler to prove than the implicit version.

We factor the difficulties in the bisimulation lemma, and prove the implicit versions from the explicit ones.

Properties: soundness and strong normalization

We prove soundness with the usual scheme: preservation + progress.

preservation: reduction preserves well-typedness

progress: errors are not well-typed

Since explicit terms represent typing derivations, the explicit version of these two lemmas is much simpler to prove than the implicit version.

We factor the difficulties in the bisimulation lemma, and prove the implicit versions from the explicit ones.

Strong normalization is proved by reification into System F.

System F_ι^p (POPL 2012)

Erasable feature	F	F_η	MLF	$\supsetneq F_{<:}$	F_ι^p
Polymorphism	$\surd$	$\surd$	-	-	$\surd$
Eta-expansion	-	$\surd$	-	$\surd$	$\surd$
Bottom	-	-	$\surd$	-	$\surd$
Top	-	-	-	$\surd$	$\surd$
Lower bounded polymorphism	-	-	$\surd$	-	$\surd$
Upper bounded polymorphism	-	-	-	$\surd$	$\surd$

We have a general framework with

- a clear distinction between programs and type annotations,
- *composable modular* coercion rules (for all erasable features),
- the preservation, progress, and bisimulation properties,
- strongly normalizing, and
- generalizing MLF, $F_{<:}$, and F_η (thus ML and F)

Part II

In this general approach to coercions, System F_ι^p seems a local optimal (with respect to coercion abstraction) that is syntactical.

Part II

In this general approach to coercions, System F_ι^p seems a local optimal (with respect to coercion abstraction) that is syntactical.

System F_ι^p can only model bounded polymorphism with a single bound, which does not even cover subtyping constraints in ML.

Part II

In this general approach to coercions, System F_ι^p seems a local optimal (with respect to coercion abstraction) that is syntactical.

System F_ι^p can only model bounded polymorphism with a single bound, which does not even cover subtyping constraints in ML.

We now lift this restriction following a semantical approach.

Coherent polymorphism

We get general coercion abstraction by simultaneously abstracting over types and coercions.

Coherent polymorphism

We get general coercion abstraction by simultaneously abstracting over types and coercions.
We factor this operation with coherent polymorphic types $\forall(\alpha : \kappa)\,\tau$ using kinds and propositions.

$$\begin{array}{rcll} \kappa & ::= & 1 \mid \kappa \times \kappa \mid \star \mid \{\alpha : \kappa \mid \mathrm{P}\} & \text{Kinds} \\ \mathrm{P} & ::= & \top \mid \mathrm{P} \wedge \mathrm{P} \mid (\Sigma \vdash \tau) \triangleright \tau & \text{Propositions} \end{array}$$

Coherent polymorphism

We get general coercion abstraction by simultaneously abstracting over types and coercions.
We factor this operation with coherent polymorphic types $\forall(\alpha : \kappa)\,\tau$ using kinds and propositions.

$\kappa ::= 1 \mid \kappa \times \kappa \mid \star \mid \{\alpha : \kappa \mid \mathrm{P}\}$	Kinds
$\mathrm{P} ::= \top \mid \mathrm{P} \wedge \mathrm{P} \mid (\Sigma \vdash \tau) \triangleright \tau$	Propositions

Examples

System F:	$\forall\alpha\,\rho$ becomes $\forall(\alpha : \star)\,\rho$
MLF:	$\forall(\alpha \triangleleft \tau)\,\rho$ becomes $\forall(\alpha : \{\beta : \star \mid \tau \triangleright \beta\})\,\rho$
Constraint ML:	$\forall(\alpha : \{\beta : \star \mid \tau_1 \triangleright \beta \wedge \beta \triangleright \tau_2\})\,\rho$ is new
	$\forall(\alpha : \{\beta : \star \times \star \mid \pi_1\,\beta \triangleright (\pi_1\,\beta) \to (\pi_2\,\beta)\})\,\rho$ is new

Why coherence?

We write $\tau \equiv \rho$ for erasable isomorphisms. It desugars to $\tau \triangleright \rho \wedge \rho \triangleright \tau$.

Why coherence?

We write $\tau \equiv \rho$ for erasable isomorphisms. It desugars to $\tau \triangleright \rho \wedge \rho \triangleright \tau$.

Abstracting over

- $\{\alpha : \star \mid (\alpha \equiv \alpha \to \alpha) \wedge (\alpha \equiv \alpha \times \alpha)\}$ looses soundness (which we must forbid)

Why coherence?

We write $\tau \equiv \rho$ for erasable isomorphisms. It desugars to $\tau \triangleright \rho \wedge \rho \triangleright \tau$.

Abstracting over

- $\{\alpha : \star \mid (\alpha \equiv \alpha \to \alpha) \wedge (\alpha \equiv \alpha \times \alpha)\}$ looses soundness (which we must forbid)
- $\{\alpha : \star \mid \alpha \equiv \alpha \to \alpha\}$ looses normalization (which we may want to forbid)

Why coherence?

We write $\tau \equiv \rho$ for erasable isomorphisms. It desugars to $\tau \triangleright \rho \wedge \rho \triangleright \tau$.

Abstracting over

- $\{\alpha : \star \mid (\alpha \equiv \alpha \to \alpha) \wedge (\alpha \equiv \alpha \times \alpha)\}$ looses soundness (which we must forbid)
- $\{\alpha : \star \mid \alpha \equiv \alpha \to \alpha\}$ looses normalization (which we may want to forbid)

Coherence

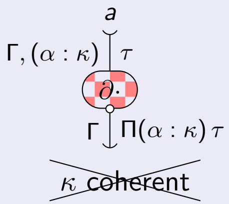
A type abstraction over κ is coherent if there is a witness ρ of kind κ. (if κ is $\{\alpha : \star \mid \alpha \triangleright \tau\}$, this implies the existence of a coercion of type $\rho \triangleright \tau$)

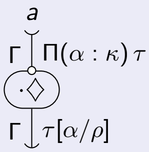
Notice that coherence holds by construction in System F_ι^p (witnesses are either top or bottom).
Coherence is a premise of the type abstraction rule in System F_{cc} (it is undecidable in the general case).

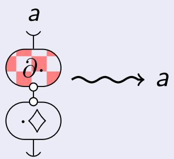
Incoherent polymorphism

Some type system features (such as GADTs) rely on blocking constructs. To study them, we extend our calculus with simple blocking and unblocking constructs.

Blocks (typing rules and reduction rule)







This construct is similar to the zero-cost abstraction of FC (Peyton Jones, Vytiniotis, 2011).

Note: This construct breaks confluence, but confluence can easily be restored.

System F_{cc}

We have a general framework with

- a clear distinction between programs and annotations,
- composable first-class coercion propositions,
- a quite rich logic (quantifiers and coinduction),
- soundness and strong normalization, and
- generalizing F_ι^p and Constraint ML (thus MLF, $\mathsf{F}_{<:}$, F_η, ML, and F)

Features not presented in this talk

Computational features (sum, unit, and void types) are easy.

Erasable features

Type language:

- top and bottom types (easy)
- equi-recursive types (requires step-indexed semantics)
- (partially done) function types as in F_ω

Proposition language:

- polymorphic propositions (easy)
- existential propositions (derives from coherence and constrained kind)
- coinduction (requires step-indexed semantics)

Formalization

The soundness and normalization results are shown using a semantic approach. Both proofs are similar, largely shared, and formalized in Coq.

Techniques

- normalization uses reducibility candidates (Tait's method)
- soundness generalizes step-indexed techniques for recursive types (Appel, McAllester, 2001) to strong reduction

Future work

There is little work to have a complete kernel type system:

- Redesign function types
- Verify that side effects can be added (with value restriction)

Future work

There is little work to have a complete kernel type system:

- Redesign function types
- Verify that side effects can be added (with value restriction)

Some interesting research remain to be done:

- Primitive existential types (call-by-constructor reduction)
- Higher-order recursive types
- Which restrictions of System F_{cc} would allow a syntactical approach?
- Intersection types need multi-premises erasable rules (partially explored, design parallel to polymorphism)
- Dependent type version of the framework?

Extra slides

Extra slides

Intersection types

$$\textsc{Intro}\quad \frac{\Gamma \vdash a : \tau_1 \qquad \Gamma \vdash a : \tau_2}{\Gamma \vdash a : \tau_1 \cap \tau_2} \qquad\qquad \textsc{Elim}\quad \frac{\Gamma \vdash a : \tau_1 \cap \tau_2}{\Gamma \vdash a : \tau_i}$$

Coercions $\Gamma \vdash (\Sigma \vdash \tau) \triangleright \rho$ now become $\Gamma \vdash \{\Sigma_i \vdash \tau_i\}^i \triangleright \rho$ which can be understood as $\Gamma \vdash \bigwedge_i (\Sigma_i \vdash \tau_i) \triangleright \rho$.

$$\textsc{Intro}\quad \Gamma \vdash \{\tau_1, \tau_2\} \triangleright \tau_1 \cap \tau_2 \qquad \textsc{Elim}\quad \Gamma \vdash \tau_1 \cap \tau_2 \triangleright \tau_i \qquad \textsc{Fusion}\quad \frac{\Gamma \vdash \{\tau, \ldots \tau\} \triangleright \rho}{\Gamma \vdash \tau \triangleright \rho}$$

$$\textsc{Intro}\quad \Gamma \vdash (\alpha \vdash \tau) \triangleright \forall \alpha\, \tau \qquad \textsc{Elim}\quad \Gamma \vdash \forall \alpha\, \tau \triangleright \tau[\alpha/\rho] \qquad \textsc{Weak}\quad \frac{\Gamma \vdash (\Sigma \vdash \tau) \triangleright \rho}{\Gamma \vdash \tau \triangleright \rho}$$

Intersection types: terms

The term typing rules are done point-wise and written $a : \{\Gamma_i \vdash \tau_i\}^i$

$$\textsc{TermVar}\quad x : \{\Gamma_i \vdash \Gamma_i(x)\}^i$$

$$\textsc{TermLam}\quad \frac{a : \{\Gamma_i, (x : \tau_i) \vdash \rho_i\}^i}{\lambda x\, a : \{\Gamma_i \vdash \tau_i \to \rho_i\}^i}$$

$$\textsc{TermApp}\quad \frac{a : \{\Gamma_i \vdash \tau_i \to \rho_i\}^i \qquad b : \{\Gamma_i \vdash \tau_i\}^i}{a\, b : \{\Gamma_i \vdash \rho_i\}^i}$$

$$\textsc{TermCoer}\quad \frac{a : \{\, \{\Gamma_i, \Sigma_j \vdash \tau_j\}^j \,\}^i \qquad \{\Gamma_i \vdash \{\Sigma_j \vdash \tau_j\}^j \triangleright \rho_i\}^i}{a : \{\Gamma_i \vdash \rho_i\}^i}$$

Intersection types: coercions

$$\textsc{Trans}\quad \frac{\left\{\Gamma, \Sigma_{2i} \vdash \{\Sigma_{1j} \vdash \tau_{1j}\}^{j} \triangleright \tau_{2i}\right\}^{i} \qquad \Gamma \vdash \{\Sigma_{2i} \vdash \tau_{2i}\}^{i} \triangleright \tau_3}{\Gamma \vdash \left\{\{\Sigma_{2i}, \Sigma_{1j} \vdash \tau_{1j}\}^{j}\right\}^{i} \triangleright \tau_3}$$

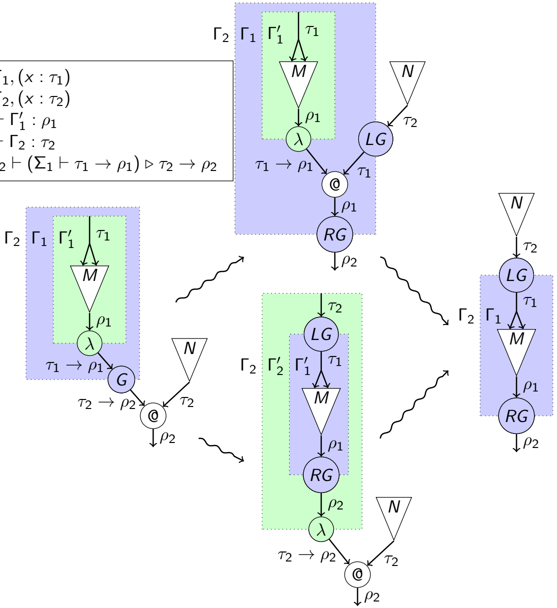
$$\textsc{Weak}\quad \frac{\Gamma \vdash \{\Sigma \vdash \tau, \ldots\} \triangleright \rho}{\Gamma \vdash \tau \triangleright \rho} \qquad \textsc{EtaArr}\quad \frac{\{\Gamma, \Sigma_i \vdash \tau' \triangleright \tau_i\}^{i} \qquad \Gamma \vdash \{\Sigma_i \vdash \rho_i\}^{i} \triangleright \rho'}{\Gamma \vdash \{\Sigma_i \vdash \tau_i \to \rho_i\}^{i} \triangleright \tau' \to \rho'}$$

We derive the usual subtyping rules:

$$\frac{\Gamma \vdash \rho \triangleright \tau_1 \qquad \Gamma \vdash \rho \triangleright \tau_2}{\Gamma \vdash \rho \triangleright \tau_1 \cap \tau_2} \qquad \Gamma \vdash (\tau \to \rho_1) \cap (\tau \to \rho_2) \triangleright \tau \to \rho_1 \cap \rho_2$$

Push

$\Gamma_1' = \Gamma_1, (x : \tau_1)$
$\Gamma_2' = \Gamma_2, (x : \tau_2)$
$_M \Rightarrow a \vdash \Gamma_1' : \rho_1$
$_N \Rightarrow b \vdash \Gamma_2 : \tau_2$
$_G \Rightarrow \Gamma_2 \vdash (\Sigma_1 \vdash \tau_1 \to \rho_1) \triangleright \tau_2 \to \rho_2$



GADTs

Existentials

coherent

$$\exists(\alpha : \kappa)\, \tau \overset{\text{def}}{=} \forall\beta\, (\forall(\alpha : \kappa)\, (\tau \to \beta)) \to \beta$$

$$\text{pack}\ a \overset{\text{def}}{=} \lambda x\ x\ a \qquad \text{unpack}\ a\ \text{as}\ x\ \text{in}\ b \overset{\text{def}}{=} a\,(\lambda x\ b)$$

incoherent

$$\Sigma(\alpha : \kappa)\, \tau \overset{\text{def}}{=} \forall\beta\, (\Pi(\alpha : \kappa)\, (\tau \to \beta)) \to \beta$$

$$\text{ipack}\ a \overset{\text{def}}{=} \lambda x\ x \diamondsuit a \qquad \text{iunpack}\ a\ \text{as}\ x\ \text{in}\ b \overset{\text{def}}{=} a\,(\partial\, \lambda x\ b)$$

$$\begin{aligned} \text{Term}\ \alpha &\overset{\text{def}}{=} \Sigma(\beta_1, \beta_2 \mid \alpha \equiv (\beta_1 \to \beta_2))\, \alpha \\ &+ \exists\beta\, \text{Term}\,(\beta \to \alpha) \times \text{Term}\ \beta \end{aligned}$$

$$\begin{aligned} \text{Lam}\ x &\overset{\text{def}}{=} \text{inl}\,(\text{ipack}\ x) &&: \forall\alpha\,\forall\beta\, (\alpha \to \beta) \to \text{Term}\,(\alpha \to \beta) \\ \text{App}\ y\ x &\overset{\text{def}}{=} \text{inr}\,(\text{pack}\ \langle y, x\rangle) &&: \forall\alpha\,\forall\beta\, \text{Term}\,(\alpha \to \beta) \to \text{Term}\ \alpha \to \text{Term}\ \beta \end{aligned}$$

$$\begin{aligned} \text{eval}\ x = \text{case}\ x\ \text{of}\ \{&\text{inl}\ x_1 \mapsto \text{iunpack}\ x_1\ \text{as}\ y\ \text{in}\ y \\ \mid\ &\text{inr}\ x_2 \mapsto \text{unpack}\ x_2\ \text{as}\ y\ \text{in}\ (\text{eval}\,(\pi_1\, y))\,(\text{eval}\,(\pi_2\, y))\} \end{aligned}$$

Why block abstraction?

We studied an alternate version of F_{cc} where type and coercion abstraction are separate constructs. Namely,

- the usual type abstraction $\forall\alpha\,\tau$ and
- the coercion abstraction $(\tau_1 \triangleright \tau_2) \Rightarrow \rho$.

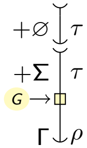
However, the test of coherence has to be done by block since some coercions hold only for some particular instances of the context.

The additional flexibility we get is thus negligible, since they are just η-expansion variants.

Weakening

To simplify readability, we add a proof of coercion weakening.

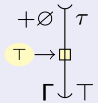
$$\dfrac{\textsc{Weak} \\ G \Rightarrow \Gamma \vdash (\Sigma \vdash \tau) \triangleright \rho}{*G \Rightarrow \Gamma \vdash \tau \triangleright \rho}$$



Extremes

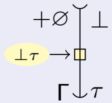
Top and bottom permit to illustrate subtyping.

Coercion rules



Top

$\top \Rightarrow \Gamma \vdash \tau \rhd \top$



Bot

$\bot_\tau \Rightarrow \Gamma \vdash \bot \rhd \tau$

Existentials

Existential types are inherently difficult (breaks preservation).

$$(x_1\, x_2 : \exists\alpha\, (\alpha \to \mathsf{Int}) \times \alpha) \vdash \mathsf{unpack}\ \alpha, x = b\ x_1\ x_2\ \mathsf{in}\ (\pi_1\, x)\, (\pi_2\, x) : \mathsf{Int}$$

erases to

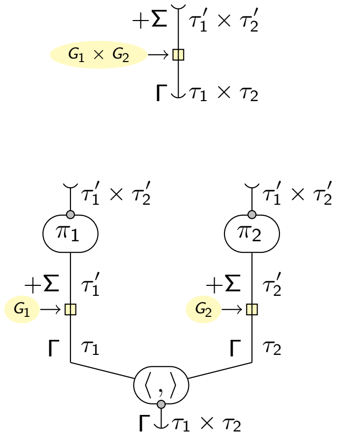
$$\mathsf{let}\ x = b\, x_1\, x_2\ \mathsf{in}\ (\pi_1\, x)\, (\pi_2\, x) \ \rightsquigarrow\ (\pi_1\, (b\, x_1\, x_2))\, (\pi_2\, (b\, x_1\, x_2))$$

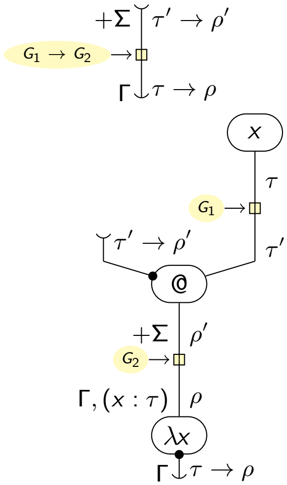
There is at least four solutions:

1. keep the calculus and use dependent types,
2. keep the calculus and use a standardization argument (Riba 2007),
3. change the calculus and use a simple argument, or
4. use the CPS encoding.

Only the second solution does not restore preservation.

Eta-expansion





Eta-expansion: distributivity

$$+\,\alpha\;\dfrac{\dfrac{\forall\alpha\,\tau \to \rho}{\tau \to \rho}\;[\alpha]}{(\forall\alpha\,\tau) \to \forall\alpha\,\rho}\;[\alpha] \to \Lambda\alpha$$

$$\frac{\Lambda\alpha \;\Rightarrow \Gamma \vdash (\,\alpha\, \vdash \rho) \triangleright \forall\alpha\,\rho \qquad [\alpha] \;\Rightarrow \Gamma,\,\alpha\, \vdash \forall\alpha\,\tau \triangleright \tau}{[\alpha] \to \Lambda\alpha \;\Rightarrow \Gamma \vdash (\,\alpha\, \vdash \tau \to \rho) \triangleright (\forall\alpha\,\tau) \to \forall\alpha\,\rho}$$

Recursive types

To subsume Constraint ML, we need equi-recursive types.

We extend

- types with recursive types $\mu\alpha\,\tau$,
- coercions with folding and unfolding, and
- propositions with coinduction.

The two rules of equi-recursive types (Amadio, Cardelli, 1993) are admissible:

$$\frac{\alpha \mapsto \rho : \mathsf{WF} \qquad \Sigma \vdash \tau_1 \equiv \rho[\alpha/\tau_1] \qquad \Sigma \vdash \rho[\alpha/\tau_2] \equiv \tau_2}{\Sigma \vdash \tau_1 \equiv \tau_2}\ \textsc{EquivPeriod}$$

$$\frac{\Sigma, (\alpha : \star, \beta : \star \mid \alpha \triangleright \beta) \vdash \tau \triangleright \rho}{\Sigma \vdash \mu\alpha\,\tau \triangleright \mu\beta\,\rho}\ \textsc{EtaMu}$$

Type functions

It is possible to extend the type system with arrow kinds. The result kind may mention the argument variable.

$$\kappa \ ::= \ \star \mid 1 \mid \kappa \times \kappa \mid (\alpha : \kappa) \to \kappa \mid \{\alpha : \kappa \mid \mathrm{P}\}$$ Kinds

The design of this extension is not satisfactory yet.

Subterm property

We will only consider erasable type system features for type systems satisfying the subterm property.

Subterm property

A type system satisfies the subterm property when the term judgment premises of its term typing rules only consider subterms of the conclusion.

OK
$$\frac{\Gamma \vdash \tau \text{ type} \qquad \Gamma, (x : \tau) \vdash a : \rho}{\Gamma \vdash \lambda x\, a : \tau \to \rho}$$

OK
$$\frac{\Gamma, \alpha \vdash a : \tau}{\Gamma \vdash a : \forall \alpha\, \tau}$$

BAD
$$\frac{x \notin \mathsf{fv}(a) \qquad \Gamma \vdash \lambda x\, a\, x : \tau \to \rho}{\Gamma \vdash a : \tau \to \rho}$$

Type systems satisfying the subterm property annotate the exact program we are interested in, not an equivalent one.

Comparison with FC

FC has only type equality coercions:

- coercions do not bind
- coercions are structural (head type constructors are equal)
  $(\forall\alpha\,\tau \to \rho) = (\tau \to \forall\alpha\,\rho)$ not a coercion

- coercions can be decomposed
- subject reduction (and bisimulation) holds

FC has coherent coercion abstraction only at top-level:

- coercion abstractions and applications cannot be fully erased

FC has a kind equality and an heterogeneous type equality.

Example with subtyping

Consequence of extending subtyping to first-class coercions:

List module signature

$\exists(\mathsf{list} : \star \to \star \mid \forall(\alpha, \beta \mid \alpha \triangleright \beta)\, \mathsf{list}\, \alpha \triangleright \mathsf{list}\, \beta)\ \ldots$

The notion of variance comes by construction and is not built-in. As such, it can be extended.