Consistent coercion calculi, and Full reduction in the face of absurdity

Julien Crétin, **Gabriel Scherer**, Didier Rémy

Gallium – INRIA

Julien Crétin was a PhD student of Didier Rémy between 2010 and 2014.

They produced a beautiful and interesting family of type systems:

Consistent coercion calculi

There was one aspect of Julien's PhD that we were not satisfied with.
The last section is about later work I did with Didier to fix it.

Section 1

Motivation and approach

Language design is hard.

Many different needs

Many different solutions

Some needs are in tensions with each other

Compromises to make, explain, and justify

Very large space of choices

How do we even know we're doing good?

How do we *test* a proposed design?

"Taste" (You only know when you *don't* have it)

How do we *test* a proposed design?

- "Taste" (You only know when you *don't* have it)
- Use cases, playing with examples, peer feedback

(idea for a conference!)

How do we *test* a proposed design?

- "Taste" (You only know when you *don't* have it)
- Use cases, playing with examples, peer feedback (idea for a conference!)
- Empirical user studies

How do we *test* a proposed design?

- "Taste" (You only know when you *don't* have it)
- Use cases, playing with examples, peer feedback

  (idea for a conference!)
- Empirical user studies
- Benchmarks (on measurable aspects)

How do we *test* a proposed design?

- “Taste” (You only know when you *don’t* have it)
- Use cases, playing with examples, peer feedback
  (idea for a conference!)
- Empirical user studies
- Benchmarks (on measurable aspects)
- Meta-theory

Meta-theory as a pre-flight checklist

Untyped:

- determinism
- confluence
- ?

Typed:

- type soundness (duh)
- weakening / monotonicity
- substitution principle / separate compilation (robustness to abstraction)
- principal types
- coherence of subtyping or type-classes

It's ok to fail a test *if you understand why*.

In this talk

Two lesser-known related tests.

Can you separate *computational* from *erasable* typing rules, and *compose* the latter?

Is your language sound for full reduction?

Test-based synthesis... for language design

Instead of testing existing languages,

Define *general* typing rules from those tests,

See what it *forces* to change in the language.

Results

A mixed bag of (seemingly unrelated) ideas:

- composing polymorphism and subtyping
- a new perspective on GADTs
- some ideas to understand erasure from proof assistants

(Maybe: Gradual typing? Contracts?)

Subsection 1

Computational and erasable rules

If you don't separate...

$$\textbf{let}\ \ li\ = \textbf{ref}\ \ [];;$$
$$li\ := 3\ ::\ \ !\,li\ ;;$$
$$li\ := \text{"foo"}\ ::\ \ !\,li\ ;;$$

$\Rightarrow$ value restriction

$$\textbf{let}\ \ li\ = \Lambda\alpha.\ \textbf{ref}\ ([] : \alpha\ \texttt{list});;$$
$$li\ \ [\,int\,]\ := 3\ ::\ \ !(\ li\ \ [\,int\ ]);;$$
$$li\ \ [bool]\ := \text{"foo"}\ ::\ \ !(\ li\ \ [\,string\ ]);;$$

not what we want

Or Haskell's monomorphic instances issue.

When you try to compose...

How do we mix polymorphism and subtyping?

F_η (Mitchell, 1988)

the natural notion of subtyping arising from polymorphism

$(\sigma \to \forall\alpha.\tau) \;\leq\; \forall\alpha.(\sigma \to \tau)$

When you try to compose...

How do we mix polymorphism and subtyping?

F_η (Mitchell, 1988)
: the natural notion of subtyping arising from polymorphism
: $(\sigma \to \forall\alpha.\tau) \;\leq\; \forall\alpha.(\sigma \to \tau)$

$F_{<:}$ (kernel, full, rec)
: the natural notion of polymorphism to add to a subtyping system
: $\forall(\alpha \leq \sigma) \ldots$

When you try to compose...

How do we mix polymorphism and subtyping?

F_η (Mitchell, 1988)
: the natural notion of subtyping arising from polymorphism
: $(\sigma \to \forall\alpha.\tau) \;\leq\; \forall\alpha.(\sigma \to \tau)$

$F_{<:}$ (kernel, full, rec)
: the natural notion of polymorphism to add to a subtyping system
: $\forall(\alpha \leq \sigma)\ldots$

ML languages in practice
: subtyping only at the toplevel
: `type t = private int`

When you try to compose...

How do we mix polymorphism and subtyping?

F_η (Mitchell, 1988)

the natural notion of subtyping arising from polymorphism

$(\sigma \to \forall\alpha.\tau) \ \leq\ \forall\alpha.(\sigma \to \tau)$

$F_{<:}$ (kernel, full, rec)

the natural notion of polymorphism to add to a subtyping system

$\forall(\alpha \leq \sigma)\ldots$

ML languages in practice

subtyping only at the toplevel

`type t = private int`

MLF

bounded abstraction motivated by principal types

$\forall(\alpha \geq \sigma)\ldots$

Subsection 2

Soundness wrt. full reduction

"Well-typed programs do not go wrong"

Closed, well-typed terms never reduce to an error.

$$\emptyset \vdash a : \tau \quad \Longrightarrow \quad \forall b,\ a \longrightarrow b,\ b \notin \mathcal{E}$$

(π_1 `true`) would raise a dynamic error,
and it is not well-typed (in OCaml, `fst true`).

Is this the only point of type systems?

"Well-typed programs do not go wrong"

Closed, well-typed terms never reduce to an error.

$$\emptyset \vdash a : \tau \quad \Longrightarrow \quad \forall b,\ a \longrightarrow b,\ b \notin \mathcal{E}$$

$(\pi_1\,\mathtt{true})$ would raise a dynamic error,
and it is not well-typed (in OCaml, `fst true`).

Is this the only point of type systems?

$$\lambda(x)\,(\pi_1\,\mathtt{true})$$

This closed term cannot evaluate to an error.
Should we improve our type systems to accept it?

$$\lambda(x)\,(\pi_1\,\texttt{true})$$

Our position: type errors are wrong even in not-yet-used parts of a program.

$$\lambda(x)\,(\pi_1\,\texttt{true})$$

Our position: type errors are wrong even in not-yet-used parts of a program.

We propose using *full reduction* when designing programming languages. Try to evaluate *open* subterms, even under λ.

"Well-typed program *fragments* do not go wrong."

$$\lambda(x)\,(\pi_1\,\mathtt{true})$$

Our position: type errors are wrong even in not-yet-used parts of a program.

We propose using *full reduction* when designing programming languages. Try to evaluate *open* subterms, even under λ.

“Well-typed program *fragments* do not go wrong.”

You have been spoiled by decades of language sound for full reduction (ML, System F)...

$$\lambda(x)\,(\pi_1\,\texttt{true})$$

Our position: type errors are wrong even in not-yet-used parts of a program.

We propose using *full reduction* when designing programming languages. Try to evaluate *open* subterms, even under λ.

"Well-typed program *fragments* do not go wrong."

You have been spoiled by decades of language sound for full reduction (ML, System F)...
until they are not anymore (GADTs!)

```
type _ tag =
  | TInt : int tag
  | TFloat : float tag

let rec double (type a) (x : a) (tag : a tag) : a =
  match tag with
  | TInt → x + x           (* assume a=int *)
  | TFloat → x +. x        (* assume a=float *)
```

```
type _ tag =
  | TInt : int tag
  | TFloat : float tag

let rec double (type a) (x : a) (tag : a tag) : a =
  match tag with
  | TInt → x + x              (* assume a=int *)
  | TFloat → x +. x           (* assume a=float *)
```

The term (double 3) has the following normal form:

```
fun tag →
  match tag with
  | TInt → 3 + 3
  | TFloat → 3 +. 3
```

Section 2

Consistent coercion calculi

$$\frac{\Gamma, x : \tau \vdash \ a \ : \sigma}{\Gamma \vdash \ \lambda(x)\, a \ : \tau \rightarrow \sigma} \qquad \frac{\Gamma \vdash \ a \ : \tau \rightarrow \sigma \qquad \Gamma \vdash \ b \ : \tau}{\Gamma \vdash \ a\ b \ : \sigma}$$

$$\frac{\Gamma, x : \tau \vdash a : \sigma}{\Gamma \vdash \lambda(x)\, a : \tau \to \sigma} \qquad \frac{\Gamma \vdash a : \tau \to \sigma \qquad \Gamma \vdash b : \tau}{\Gamma \vdash a\, b : \sigma}$$

$$\frac{\Gamma, \alpha \vdash a : \tau}{\Gamma \vdash a : \forall(\alpha)\, \tau} \qquad \frac{\Gamma \vdash a : \forall(\alpha)\, \tau \qquad \Gamma \vdash \sigma : \star}{\Gamma \vdash a : \tau[\sigma/\alpha]}$$

$$\frac{\Gamma, x : \tau \vdash\ a\ : \sigma}{\Gamma \vdash\ \lambda(x)\, a\ : \tau \to \sigma} \qquad \frac{\Gamma \vdash\ a\ : \tau \to \sigma \qquad \Gamma \vdash\ b\ : \tau}{\Gamma \vdash\ a\ b\ : \sigma}$$

$$\frac{\Gamma, \alpha \vdash\ a\ : \tau}{\Gamma \vdash\ a\ : \forall(\alpha)\, \tau} \qquad \frac{\Gamma \vdash\ a\ : \forall(\alpha)\, \tau \qquad \Gamma \vdash \sigma : \star}{\Gamma \vdash\ a\ : \tau[\sigma/\alpha]}$$

$$\frac{\Gamma \vdash\ a\ : \tau \qquad \Gamma \vdash\ b\ : \sigma}{\Gamma \vdash\ (a, b)\ : \tau \times \sigma} \qquad \frac{\Gamma \vdash\ a\ : \tau_1 \times \tau_2}{\Gamma \vdash\ \pi_i\, a\ : \tau_i}$$

$$\frac{\Gamma, x : \tau \vdash a : \sigma}{\Gamma \vdash \lambda(x)\, a : \tau \to \sigma} \qquad \frac{\Gamma \vdash a : \tau \to \sigma \qquad \Gamma \vdash b : \tau}{\Gamma \vdash a\, b : \sigma}$$

$$\frac{\Gamma, \alpha \vdash a : \tau}{\Gamma \vdash a : \forall(\alpha)\, \tau} \qquad \frac{\Gamma \vdash a : \forall(\alpha)\, \tau \qquad \Gamma \vdash \sigma : \star}{\Gamma \vdash a : \tau[\sigma/\alpha]}$$

$$\frac{\Gamma \vdash a : \tau \qquad \Gamma \vdash b : \sigma}{\Gamma \vdash (a, b) : \tau \times \sigma} \qquad \frac{\Gamma \vdash a : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\, a : \tau_i}$$

$$\frac{\Gamma \vdash a : \tau \qquad \Gamma \vdash \tau \leq \sigma}{\Gamma \vdash a : \sigma}$$

$$\frac{\Gamma, x : \tau \vdash \boxed{a} : \sigma}{\Gamma \vdash \boxed{\lambda(x)\, a} : \tau \to \sigma} \qquad \frac{\Gamma \vdash \boxed{a} : \tau \to \sigma \qquad \Gamma \vdash \boxed{b} : \tau}{\Gamma \vdash \boxed{a\ b} : \sigma}$$

$$\frac{\Gamma, \alpha \vdash \boxed{a} : \tau}{\Gamma \vdash \boxed{a} : \forall(\alpha)\, \tau} \qquad \frac{\Gamma \vdash \boxed{a} : \forall(\alpha)\, \tau \qquad \Gamma \vdash \sigma : \star}{\Gamma \vdash \boxed{a} : \tau[\sigma/\alpha]}$$

$$\frac{\Gamma \vdash \boxed{a} : \tau \qquad \Gamma \vdash \boxed{b} : \sigma}{\Gamma \vdash \boxed{(a, b)} : \tau \times \sigma} \qquad \frac{\Gamma \vdash \boxed{a} : \tau_1 \times \tau_2}{\Gamma \vdash \boxed{\pi_i\, a} : \tau_i}$$

$$\frac{\Gamma \vdash \boxed{a} : \tau \qquad \Gamma \vdash \tau \leq \sigma}{\Gamma \vdash \boxed{a} : \sigma}$$

$$\frac{\Gamma, x : \tau \vdash \boxed{a} : \sigma}{\Gamma \vdash \boxed{\lambda(x)\, a} : \tau \to \sigma} \qquad \frac{\Gamma \vdash \boxed{a} : \tau \to \sigma \qquad \Gamma \vdash \boxed{b} : \tau}{\Gamma \vdash \boxed{a\ b} : \sigma}$$

$$\frac{\Gamma, \alpha \vdash \boxed{a} : \tau}{\Gamma \vdash \boxed{a} : \forall(\alpha)\, \tau} \qquad \frac{\Gamma \vdash \boxed{a} : \forall(\alpha)\, \tau \qquad \Gamma \vdash \sigma : \star}{\Gamma \vdash \boxed{a} : \tau[\sigma/\alpha]}$$

$$\frac{\Gamma \vdash \boxed{a} : \tau \qquad \Gamma \vdash \boxed{b} : \sigma}{\Gamma \vdash \boxed{(a, b)} : \tau \times \sigma} \qquad \frac{\Gamma \vdash \boxed{a} : \tau_1 \times \tau_2}{\Gamma \vdash \boxed{\pi_i\, a} : \tau_i}$$

$$\frac{\Gamma \vdash \boxed{a} : \tau \qquad \Gamma \vdash \tau \leq \sigma}{\Gamma \vdash \boxed{a} : \sigma}$$

Computational and Erasable rules:

$$\frac{(\Gamma, \Delta_i \vdash \boxed{a_i} : \tau_i)^{i \in 1 \ldots n} \qquad J_1 \ldots J_m}{\Gamma \vdash \boxed{node(a_1, \ldots, a_n)} : \sigma} \qquad \frac{\Gamma, \Delta \vdash \boxed{a} : \tau \qquad J_1 \ldots J_m}{\Gamma \vdash \boxed{a} : \sigma}$$

Introducing coercions

We had many erasable rules:

$$\frac{\Gamma, \Delta \vdash a : \tau \qquad J_1 \ldots J_m}{\Gamma \vdash a : \sigma}$$

Factor all erasable rules into a unique term rule,
and coercion rules:

$$\textsc{TermCoer}\ \frac{\Gamma, \Delta \vdash a : \tau \qquad \Gamma \vdash (\Delta \vdash \tau) \triangleright \sigma}{\Gamma \vdash a : \sigma} \qquad \frac{J_1 \ldots J_m}{\Gamma \vdash (\Delta \vdash \tau) \triangleright \sigma}$$

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.

$$\frac{J_1 \ldots J_m}{\Gamma \vdash (\Delta \vdash \tau) \triangleright \sigma}$$

More importantly,

- Rule TermCoer enforces a unique interface for all erasable features
  - $(\Delta \vdash \tau) \triangleright \sigma$
  - "typing coercion" (converting whole judgments)
- This change is a preliminary to have composable features.
- It requires decomposing existing features into atomic parts.

Consistent coercion calculi (Ccc)

$$
\begin{array}{llll}
\tau, \sigma & ::= & \ldots & \text{types} \\
\kappa & ::= & \ldots & \text{kinds} \\
P, Q & ::= & \ldots & \text{propositions} \\
\Gamma, \Delta & ::= & \emptyset \mid \Gamma, x : \tau \mid \Gamma, \alpha : \kappa \mid \Gamma, \phi : P & \text{environments}
\end{array}
$$

Judgments:

$$
\begin{array}{ll}
\Gamma \vdash a : \tau & \text{terms} \\
\Gamma \vdash \tau : \kappa & \text{types} \\
\Gamma \vdash P \ \texttt{true} & \text{propositions} \\
\Gamma \vdash (\Delta \vdash \tau) \triangleright \sigma & \text{coercions}
\end{array}
$$

Ccc: term typing rules

$$
\begin{array}{lcl}
\tau, \sigma & ::= & \alpha, \beta, \gamma \cdots \mid \tau \to \sigma \mid \tau \times \sigma \\
\kappa & ::= & \star
\end{array}
$$

$$\frac{\Gamma, x : \tau \vdash a : \sigma}{\Gamma \vdash \lambda(x)\, a : \tau \to \sigma} \qquad \frac{\Gamma \vdash a : \tau \to \sigma \qquad \Gamma \vdash b : \tau}{\Gamma \vdash a\ b : \sigma}$$

$$\frac{\Gamma \vdash a : \tau \qquad \Gamma \vdash b : \sigma}{\Gamma \vdash (a, b) : \tau \times \sigma} \qquad \frac{\Gamma \vdash a : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\, a : \tau_i}$$

$$\frac{\Gamma, \Delta \vdash a : \tau \qquad \Gamma \vdash (\Delta \vdash \tau) \triangleright \sigma}{\Gamma \vdash a : \sigma}\ \textsc{TermCoerce}$$

(easily encode subtyping: $(\tau \triangleright \sigma) := (\emptyset \vdash \tau) \triangleright \sigma$)

Feature: polymorphism

Old rules:

$$\frac{\Gamma, \alpha : \kappa \vdash a : \tau}{\Gamma \vdash a : \forall(\alpha : \kappa)\, \tau}\ \textsc{TermGen} \qquad \frac{\Gamma \vdash a : \forall(\alpha : \kappa)\, \tau \qquad \Gamma \vdash \sigma : \kappa}{\Gamma \vdash a : \tau[\sigma/\alpha]}\ \textsc{TermInst}$$

In Ccc:

$$\tau, \sigma \quad ::= \quad \cdots \mid \forall(\alpha : \kappa)\, \tau$$

$$\frac{?}{\Gamma \vdash ((\alpha : \kappa) \vdash \tau) \triangleright \forall(\alpha : \kappa)\, \tau}\ \textsc{CoerGen?} \qquad \frac{\Gamma \vdash \sigma : \kappa}{\Gamma \vdash (\emptyset \vdash \forall(\alpha : \kappa)\, \tau) \triangleright \tau[\sigma/\alpha]}\ \textsc{CoerInst}$$

Ccc: propositions

Propositions is the logical layer in which the various side-conditions of the judgments live. Light reasoning power.

$$
\begin{array}{llll}
P, Q & ::= & & \text{propositions} \\
& \mid & P \wedge Q & \text{conjunction} \\
& \mid & \top & \text{true} \\
& & & \\
& \mid & (\Delta \vdash \tau) \triangleright \sigma & \text{coercions} \\
& \mid & \ldots & \text{and other primitive notions...}
\end{array}
$$

$$
\frac{\Gamma \vdash P\ \texttt{true} \qquad \Gamma \vdash Q\ \texttt{true}}{\Gamma \vdash (P \wedge Q)\ \texttt{true}}
\qquad
\frac{\Gamma \vdash (P_1 \wedge P_2)\ \texttt{true}}{\Gamma \vdash P_i\ \texttt{true}}
\qquad
\Gamma \vdash \top\ \texttt{true}
$$

$$
\frac{\Gamma \vdash (\Delta \vdash \tau) \triangleright \sigma}{\Gamma \vdash ((\Delta \vdash \tau) \triangleright \sigma)\ \texttt{true}}
$$

Feature: bounded polymorphism

The most general way to allow abstractions $\forall(\alpha \le \sigma)\ \dots$ is to use *refinement kinds* $\{\alpha : \kappa \mid P\}$.

$$\kappa \quad ::= \quad \cdots \mid \{\alpha : \kappa \mid P\}$$

$$\frac{\Gamma \vdash \tau : \kappa \qquad \Gamma \vdash P[\tau/\alpha]}{\Gamma \vdash \tau : \{\alpha : \kappa \mid P\}} \qquad \frac{\Gamma \vdash \tau : \{\alpha : \kappa \mid P\}}{\Gamma \vdash \tau : \kappa} \qquad \frac{\Gamma \vdash \tau : \{\alpha : \kappa \mid P\}}{\Gamma \vdash \tau : P[\tau/\alpha]}$$

Then $\forall(\alpha \le \sigma)\ \dots$ is expressible as...

$$\forall(\alpha : \{\alpha : \star \mid (\emptyset \vdash \alpha) \triangleright \sigma\})\ \dots$$

Le loup dans la bergerie

Problem: unrestricted erasable polymorphism is unsound.

Le loup dans la bergerie

Problem: unrestricted erasable polymorphism is unsound.

$$\dfrac{\dfrac{\dfrac{\vdash \texttt{true} : \mathbb{B} \qquad \alpha : \{\alpha : \star \mid \mathbb{B} \triangleright \mathbb{B} \times \mathbb{B}\} \vdash \mathbb{B} \triangleright \mathbb{B} \times \mathbb{B}}{\alpha : \{\alpha : \star \mid \mathbb{B} \triangleright \mathbb{B} \times \mathbb{B}\} \vdash \texttt{true} : \mathbb{B} \times \mathbb{B}}}{\alpha : \{\alpha : \star \mid \mathbb{B} \triangleright \mathbb{B} \times \mathbb{B}\} \vdash (\pi_1\, \texttt{true}) : \mathbb{B}}}{\emptyset \vdash (\pi_1\, \texttt{true}) : \forall(\alpha : \{\alpha : \star \mid (\emptyset \vdash \mathbb{B}) \triangleright \mathbb{B} \times \mathbb{B}\})\, \mathbb{B}}$$

A well-typed program that goes wrong.

The problem is that $\mathbb{B} \triangleright \mathbb{B} \times \mathbb{B}$ is absurd – and unprovable.
We have to restrict these typing rules.

Consistency

The sound rules for polymorphism enforce *consistent* abstraction.

$$\begin{array}{lcl} \tau, \sigma & ::= & \cdots \mid \forall(\alpha : \kappa)\,\tau \\ P & ::= & \cdots \mid \exists\kappa \end{array}$$

$$\textsc{CoerGen}\quad \frac{\Gamma \vdash \exists\kappa\ \texttt{true}}{\Gamma \vdash ((\alpha : \kappa) \vdash \tau) \triangleright \forall(\alpha : \kappa)\,\tau}$$

$$\textsc{CoerInst}\quad \frac{\Gamma \vdash \sigma : \kappa}{\Gamma \vdash (\emptyset \vdash \forall(\alpha : \kappa)\,\tau) \triangleright \tau[\sigma/\alpha]}$$

$$\textsc{PropConsist}\quad \frac{\Gamma \vdash \sigma : \kappa}{\Gamma \vdash \exists\kappa\ \texttt{true}}$$

This is one contribution of Julien Crétin's PhD thesis: highlighting *consistency* as the key to erasable polymorphism.

Feature: η-expansion (depth subtyping)

$$\frac{\Gamma, \Delta \vdash (\emptyset \vdash \sigma_1) \triangleright \tau_1 \qquad \Gamma \vdash (\Delta \vdash \tau_2) \triangleright \sigma_2}{\Gamma \vdash (\Delta \vdash \tau_1 \to \tau_2) \triangleright \sigma_1 \to \sigma_2}\ \textsc{CoerArr}$$

$$\frac{\Gamma \vdash (\Delta \vdash \tau_1) \triangleright \sigma_1 \qquad \Gamma \vdash (\Delta \vdash \tau_2) \triangleright \sigma_2}{\Gamma \vdash (\Delta \vdash \tau_1 \times \tau_2) \triangleright \sigma_1 \times \sigma_2}\ \textsc{CoerProd}$$

Only one such rule needed for each type.
Distributivity rules are derivable, using...

Ccc: coercions

Structural rules:

$$\textsc{CoerRefl}\quad \Gamma \vdash \tau \rhd \tau$$

$$\textsc{CoerTrans}\quad \frac{\Gamma, \Delta_1 \vdash (\Delta_2 \vdash \tau_3) \rhd \tau_2 \qquad \Gamma \vdash (\Delta_1 \vdash \tau_2) \rhd \tau_1}{\Gamma \vdash (\Delta_1, \Delta_2 \vdash \tau_3) \rhd \tau_1}$$

$$\textsc{CoerProp}\quad \frac{\Gamma \vdash (\Delta \vdash \tau) \rhd \sigma \ \texttt{true} \qquad \Gamma \vdash \exists \Delta \ \texttt{true}}{\Gamma \vdash (\Delta \vdash \tau) \rhd \sigma}$$

With these rules, various features, once expressed as coercions, can be composed together.

Feature: multi-abstraction

$$\kappa \quad ::= \quad \cdots \mid \kappa \times \kappa$$

$$\frac{\Gamma \vdash \tau : \kappa_1 \qquad \Gamma \vdash \sigma : \kappa'}{\Gamma \vdash (\tau, \sigma) : \kappa \times \kappa'} \qquad\qquad \frac{\Gamma \vdash \tau : \kappa_1 \times \kappa_2}{\Gamma \vdash \pi_i\, \tau_i : \kappa_i}$$

Express $\forall(\alpha_1, \alpha_2 \mid \alpha_1 \leq \alpha_2 \to \alpha_1)$ with

$$\forall(\alpha : \{\alpha : \star \times \star \mid \pi_1\, \alpha \triangleright (\pi_2\, \alpha \to \pi_1\, \alpha)\})$$

Feature: recursive types and coinduction

$$\tau, \sigma \quad ::= \quad \cdots \mid \mu\alpha.\,\tau$$

$$\textsc{CoerUnfold}\quad \frac{}{\Gamma \vdash \mu\alpha.\,\tau \triangleright \tau[\mu\alpha.\,\tau/\alpha]} \qquad \textsc{CoerFold}\quad \frac{\Gamma \vdash \mu\alpha.\,\tau : \star}{\Gamma \vdash \tau[\mu\alpha.\,\tau/\alpha] \triangleright \mu\alpha.\,\tau}$$

Replace $\Gamma \vdash P$ `true` by $\Gamma; \Theta \vdash P$ `true`
to keep track of coinductive hypotheses.

$$\textsc{PropFix}\quad \frac{\Gamma; \Theta, P \vdash P}{\Gamma; \Theta \vdash P} \qquad \textsc{CoerProd'}\quad \frac{\Gamma, \Theta; \emptyset \vdash (\Delta \vdash \tau_1) \triangleright \sigma_1 \qquad \Gamma, \Theta; \emptyset \vdash (\Delta \vdash \tau_2) \triangleright \sigma_2}{\Gamma; \Theta \vdash (\Delta \vdash \tau_1 \times \tau_2) \triangleright \sigma_1 \times \sigma_2} \qquad \ldots$$

Productivity: coinductive hypotheses available under computational connectives.

The usual equi-recursive rules are *derivable*.

Properties

Soundness and termination are formalized in Coq.

Step-indexed techniques, adapted to full reduction by moving indices inside terms.

Achieved

F_η: subsumed by η-coercions.

$F_{<:}$: $\forall(\alpha \le \tau)\sigma$ is encoded as $\forall(\alpha : \{\alpha : \star \mid \alpha \triangleright \tau\})\,\sigma$.

MLF: $\forall(\alpha \ge \tau)\sigma$ is encoded as $\forall(\alpha : \{\alpha : \star \mid \tau \triangleright \alpha\})\,\sigma$.

ML with subtyping constraints: bag of constraints using product kinds.

All features can be combined together.

Still missing

GADTs: Based on equality constraints can be encoded $(\tau \triangleright \sigma) \wedge (\sigma \triangleright \tau)$, but also require abstraction over inconsistent kinds...

Section 3

Full reduction in presence of absurdity

Full reduction

Consistent coercion calculi as presented above are sound for *full* reduction. Easy to define thanks to erasability.

$$E \ ::= \ \square \mid \lambda(x)\, E \mid E\ b \mid a\ E \mid (E, a) \mid (a, E) \mid \pi_i\, E \qquad \text{Contexts}$$

$$(\lambda(x)\, a)\ b \circ\!\!\longrightarrow a[b/x] \qquad \pi_i\,(a_1, a_2) \circ\!\!\longrightarrow a_i \qquad \frac{a \circ\!\!\longrightarrow b}{E[a] \longrightarrow E[b]}$$

Nota Bene

To prove consistency of $\{\alpha : \kappa \mid P\}$ in context Γ, one may pick α arbitrary (it is *flexible*) but Γ is *rigid*.

$$\cfrac{\cfrac{\Gamma \vdash \sigma : \kappa \qquad \Gamma \vdash P[\sigma/\alpha]\ \texttt{true}}{\Gamma \vdash \sigma : \{\alpha : \kappa \mid P\}}}{\Gamma \vdash \exists\{\alpha : \kappa \mid P\}\ \texttt{true}}$$

Recall our previous GADT example:

```
type _ tag =
  | TInt : int tag
  | TFloat : float tag

let rec double (type a) (x : a) (tag : a tag) : a =
  match tag with
  | TInt → x + x         (* assume a=int; unsatisfiable ! *)
  | TFloat → x +. x      (* assume a=float; unsatisfiable ! *)
```

`(a = int)` is unsatisfiable when `a` is rigid.

GADTs need inconsistent abstraction.

Explicit vs. Implicit

In dependently typed languages, logical propositions are naturally represented as types. Assumptions are made using just $\lambda(x : P)\, a$.

Explicit vs. Implicit

In dependently typed languages, logical propositions are naturally represented as types. Assumptions are made using just $\lambda(x : P)\, a$.

```
(fun (type a) (x : a) (tag : a tag) → match tag with
  | TInt (w : a = int) → (w x) + (w x)
  | TFloat (w : a = float) → (w x) +. (w x)
) 3
```

If *each use* of an assumption is marked by the free variables, *all* dangerous redexes are blocked by those variables.
Same in functional intermediate typed representations (eg. System FC).

Explicit vs. Implicit

In dependently typed languages, logical propositions are naturally represented as types. Assumptions are made using just $\lambda(x : P)\, a$.

$$
\begin{array}{l}
(\textbf{fun}\ (\textbf{type}\ \mathsf{a})\ (\mathsf{x} : \mathsf{a})\ (\mathsf{tag} : \mathsf{a}\ \mathsf{tag}) \to \textbf{match}\ \mathsf{tag}\ \textbf{with} \\
\quad \mid\ \mathsf{TInt}\ (\mathsf{w} : \mathsf{a} = \mathsf{int}) \to (\mathsf{w}\ \mathsf{x}) + (\mathsf{w}\ \mathsf{x}) \\
\quad \mid\ \mathsf{TFloat}\ (\mathsf{w} : \mathsf{a} = \mathsf{float}) \to (\mathsf{w}\ \mathsf{x}) +. (\mathsf{w}\ \mathsf{x}) \\
)\ 3
\end{array}
$$

If *each use* of an assumption is marked by the free variables,
all dangerous redexes are blocked by those variables.
Same in functional intermediate typed representations (eg. System FC).

Erasability + user convenience: assumptions should be usable *implicitly* in derivations. Just as consistent abstraction.

A type $[P]$ for explicitly assuming P.
$[P]$ can be "opened", which makes the assumption implicitly usable – but it blocks computation.

$$\tau \mathrel{+}::= [P] \qquad\qquad a \mathrel{+}::= \diamond \mid \delta(a, \phi.b)$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash \diamond : [P]} \qquad\qquad \frac{\Gamma \vdash a : [P] \qquad \Gamma, \phi : P \vdash b : \tau}{\Gamma \vdash \delta(a, \phi.b) : \tau}$$

A type $[P]$ for explicitly assuming P.
$[P]$ can be "opened", which makes the assumption implicitly usable – but it blocks computation.

$$\tau \mathrel{+}::= [P] \qquad\qquad a \mathrel{+}::= \diamond \mid \delta(a, \phi.b)$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash \diamond : [P]} \qquad\qquad \frac{\Gamma \vdash a : [P] \qquad \Gamma, \phi : P \vdash b : \tau}{\Gamma \vdash \delta(a, \phi.b) : \tau}$$

$$E \mathrel{+}::= \delta(E, \phi.Q) \mid \cancel{\delta(a, \phi.E)} \qquad\qquad \delta(\diamond, \phi.b) \multimap b$$

A type $[P]$ for explicitly assuming P.
$[P]$ can be "opened", which makes the assumption implicitly usable – but it blocks computation.

$$\tau \mathrel{+}::= [P] \qquad\qquad a \mathrel{+}::= \diamond \mid \delta(a,\, \phi.b)$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash \diamond : [P]} \qquad\qquad \frac{\Gamma \vdash a : [P] \qquad \Gamma, \phi : P \vdash b : \tau}{\Gamma \vdash \delta(a,\, \phi.b) : \tau}$$

$$E \mathrel{+}::= \delta(E,\, \phi.Q) \mid \cancel{\delta(a,\, \phi.E)} \qquad\qquad \delta(\diamond,\, \phi.b) \multimap b$$

$$\emptyset \vdash \lambda(x)\, \delta(x,\, \phi.(\pi_1\, \mathtt{true})) : [\mathbb{B} \triangleright \mathbb{B} \times \mathbb{B}] \to \mathbb{B}$$

What you can block, you can un-block

$$\delta(a,\ \phi.\ E\left[\ F\left[\ b\ \right]\ \right])$$

What you can block, you can un-block

$$\delta(a,\ \phi.\ E\left[\ F\left[\ b\ \right]\ \right])$$

$$E\left[\delta(a,\ \phi.\ F\left[\ b\ \right]\ )\right]$$

What you can block, you can un-block

$$\delta(a,\ \phi.\ \boxed{E\left[\ \boxed{F\left[\ \boxed{b}\ \right]}\ \right]})$$

$$E\left[\delta(a,\ \phi.\ \boxed{F\left[\ \boxed{b}\ \right]}\ )\right]$$

For flexibility, allow un-blocking a subterm by disabling an assumption.

What you can block, you can un-block

$$\delta(a,\ \phi.\ \boxed{E\left[\boxed{F\left[\boxed{b}\right]}\right]})$$

$$E\left[\delta(a,\ \phi.\ \boxed{F\left[\boxed{b}\right]})\right]$$

For flexibility, allow un-blocking a subterm by disabling an assumption.

$$E\left[\delta(a,\ \phi.\ \boxed{F\left[\mathtt{hide}\,\phi\,\mathtt{in}\ \boxed{b}\right]})\right]$$

What you can block, you can un-block

$$\delta(a,\ \phi.\ \boxed{E\left[\boxed{F\left[\boxed{b}\right]}\right]})$$

$$E\left[\delta(a,\ \phi.\ \boxed{F\left[\boxed{b}\right]})\right]$$

For flexibility, allow un-blocking a subterm by disabling an assumption.

$$E\left[\delta(a,\ \phi.\ \boxed{F\left[\mathtt{hide}\,\phi\,\mathtt{in}\ \boxed{b}\right]})\right]$$

$$a \mathrel{+}::= \mathtt{hide}\,\phi\,\mathtt{in}\,b \qquad\qquad \frac{\Gamma \vdash \Delta \qquad \Gamma, \Delta \vdash a : \tau}{\Gamma, \phi : P, \Delta \vdash \mathtt{hide}\,\phi\,\mathtt{in}\,a : \tau}$$

Mixing full and weak reduction: confluence in danger!

Suppose $a \longrightarrow b$. We have a confluence problem

$$\begin{tikzcd}
(\lambda(x)\,\delta(y,\,\phi.\,\boxed{E[x]}))\ a \arrow[r] \arrow[d] & (\lambda(x)\,\delta(y,\,\phi.\,\boxed{E[x]}))\ b \arrow[d] \\
\delta(y,\,\phi.\,\boxed{E[\qquad a\ ]}) \arrow[r, "\not\longrightarrow" description, phantom] & \delta(y,\,\phi.\,\boxed{E\left[\qquad b\ \right]})
\end{tikzcd}$$

A term in reducible position before substitution, should remain reducible after substitution.

Mixing full and weak reduction: confluence in danger!

Suppose $a \longrightarrow b$. We have a confluence problem

$$
\begin{tikzcd}
(\lambda(x)\,\delta(y,\,\phi.\,E[x]))\;a \arrow[r] \arrow[d] & (\lambda(x)\,\delta(y,\,\phi.\,E[x]))\;b \arrow[d] \\
\delta(y,\,\phi.\,E[\qquad a\;]) \arrow[r, "/" marking] & \delta(y,\,\phi.\,E\left[\qquad b\;\right])
\end{tikzcd}
$$

A term in reducible position before substitution, should remain reducible after substitution.

Idea: insert $\texttt{hide}\,\phi$ when substitution traverses the guard ϕ.

Mixing full and weak reduction: confluence restored.

Suppose $a \longrightarrow b$. We have a confluence problem

$$\begin{array}{ccc} (\lambda(x)\,\delta(y,\,\phi.\,\boxed{E[x]}))\;a & \longrightarrow & (\lambda(x)\,\delta(y,\,\phi.\,\boxed{E[x]}))\;b \\ \downarrow & & \downarrow \\ \delta(y,\,\phi.\,\boxed{E[\mathtt{hide}\,\phi\,\mathtt{in}\,\boxed{a}\,]}) & \longrightarrow & \delta(y,\,\phi.\,\boxed{E\left[\mathtt{hide}\,\phi\,\mathtt{in}\,\boxed{b}\,\right]}) \end{array}$$

A term in reducible position before substitution, should remain reducible after substitution.

Idea: insert $\mathtt{hide}\,\phi$ when substitution traverses the guard ϕ.

The resulting system is *sound* for full-reduction and *confluent* (new proof!).

GADTs, sound edition

```
type 'a tag =
  | TInt of ['a = int]
  | TFloat of ['a = float]

let rec double (type a) (x : a) (tag : a tag) : a =
  match tag with
  | TInt w → δ(w, φ. x + x)
  | TFloat w → δ(w, φ. x +. x)
```

GADTs, sound edition

```
type 'a tag =
  | TInt of ['a = int]
  | TFloat of ['a = float]

let rec double (type a) (x : a) (tag : a tag) : a =
  match tag with
  | TInt w → δ(w, φ. x) + δ(w, φ. x)
  | TFloat w → δ(w, φ. x) +. δ(w, φ. x)
```

GADTs, sound edition

```
type 'a tag =
  | TInt of ['a = int]
  | TFloat of ['a = float]

let rec double (type a) (x : a) (tag : a tag) : a =
  match tag with
  | TInt w → δ(w, φ. x) + δ(w, φ. x)
  | TFloat w → δ(w, φ. x) +. δ(w, φ. x)
```

We offer a continuum between fully-explicit and fully-implicit use of assumptions.

Dependable ideas

We call $\Gamma \vdash P$ `true` the *definitional truth* judgment:
true by fiat.

The type $[P]$ corresponds to *propositional truths*:
evidence passed around by the user.

We allow *consistent* abstraction over definitional truths
– something not usually studied in intensional type theories.

Empty and non-empty contexts (eg., for extraction):
consistent vs. *inconsistent* contexts.

In an article near you

http://gallium.inria.fr/~scherer/drafts/consistency-draft.pdf

A novel formal proof of confluence with parallel reduction, with Wright-Felleisen separation of head redexes and contexts: scales to larger languages.

Detailed soundness proofs by bisimulations with known-sound calculi – and administrative variants thereof.

Take away

Consistent coercions allow to compose erasable type-system features.

Opinion: in many case programmers think of correctness in an abstract way, *full reduction*. Confluence is essential.

We should distinguish *consistent* and (possibly) *inconsistent* abstractions.

Unified approaches have downsides for both; language design could help preserve/structure this distinction.

To support confluent inconsistent abstraction, one must allow to block (soundness) and *unblock* (confluence) reduction of subterms.

Thanks! Questions?