Full reduction and GADTs

Didier Rémy

(Joint work with Gabriel Scherer)

INRIA - Paris

APLS, Dec 2015

Typed soundness —*Our slogan*

“ **Well-typed programs do not go wrong** ”

Typed soundness *—Our slogan*

“ **Well-typed programs do not go wrong** ”

But what does this means?

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

$$\emptyset \vdash a : \tau \quad \Longrightarrow \quad \forall b,\ a \xrightarrow{*} b,\ b \notin \mathit{Errors}$$

Typed soundness —*Our slogan*

“ **Well-typed programs do not go wrong** ”

But what does this means?

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

$$\emptyset \vdash a : \tau \quad \Longrightarrow \quad \forall b,\ a \xrightarrow{*} b,\ b \notin \mathit{Errors}$$

The term $\pi_1\, \texttt{true}$ is an error, but it is ill-typed. —*So we are happy.*

Typed soundness — *Our slogan*

“ Well-typed programs do not go wrong ”

But what does this means?

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

$$\emptyset \vdash a : \tau \quad \Longrightarrow \quad \forall b,\ a \xrightarrow{*} b,\ b \notin \mathit{Errors}$$

The term $\pi_1\ \mathtt{true}$ is an error, but it is ill-typed. —*So we are happy.*

However...

$\lambda(x)\,(\pi_1\ \mathtt{true})$ *is not an error, but it is still ill-typed.*

—*Should we be upset?*

Should we fix/improve our type system to accept this?

Typed soundness —*Our slogan revisited*

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

Type errors are bad even in not-yet-used parts of a program.

Typed soundness — *Our slogan revisited*

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

Type errors are bad even in not-yet-used parts of a program.

$$\lambda(x)\,(\lambda(y)\,\pi_1\,y)\,\texttt{true} \xrightarrow{\textit{full}} \lambda(x)\,(\pi_1\,\texttt{true})$$

Latent type errors are also bad.

Problem: These errors are not ruled out by type soundness for CBV.

Typed soundness —*Our slogan revisited*

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

Type errors are bad even in not-yet-used parts of a program.

$$\lambda(x)\,\boxed{(\lambda(y)\,\pi_1\,y)\,\texttt{true}} \xrightarrow{\textit{full}} \lambda(x)\,(\boxed{\pi_1\,\texttt{true}})$$

Latent type errors are also bad.

Problem: These errors are not ruled out by type soundness for CBV.

Solution: Full reduction should be used to test type soundness!

This will evaluate *open* subterms, even under λ's.

Revised slogan (with full reduction)

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

Benefit of full reduction

Detects more errors.

- Hence, as a corollary, type soundness is a stronger result!

Makes typechecking more modular:

- You are not forced to *use* your functions to see errors in their bodies.

Share the meta-theoretical study between CBV and CBN.

Also gives a more abstract view of programs

- Even in languages with a CBV semantics, full reduction may be used to understand programs when efficiency is not a concern.

Gives a more solid ground

- Even if a full-fledged language uses CBV, it is reassuring if its core subset is sound and confluent for full reduction.

Benefit of full reduction

Detects more errors.

- Hence, as a corollary, type soundness is a stronger result!

Makes typechecking more modular:

- You are not forced to *use* your functions to see errors in their bodies.

Share the meta-theoretical study between CBV and CBN.

Also gives a more abstract view of programs

- Even in languages with a CBV semantics, full reduction may be used to understand programs when efficiency is not a concern.

Gives a more solid ground

- Even if a full-fledged language uses CBV, it is reassuring if its core subset is sound and confluent for full reduction.

Beware! We've been spoiled by decades during which our languages were sound for full reduction and these properties could be taken for granted.

Benefit of full reduction

Detects more errors.

- Hence, as a corollary, type soundness is a stronger result!

Makes typechecking more modular:

- You are not forced to *use* your functions to see errors in their bodies.

Share the meta-theoretical study between CBV and CBN.

Also gives a more abstract view of programs

- Even in languages with a CBV semantics, full reduction may be used to understand programs when efficiency is not a concern.

Gives a more solid ground

- Even if a full-fledged language uses CBV, it is reassuring if its core subset is sound and confluent for full reduction.

Beware! We've been spoiled by decades during which our languages were sound for full reduction and these properties could be taken for granted. — **But this is not true anymore!**

Full reduction with GADTs

```
type _ tag =
  | TInt : int tag
  | TString : string tag

let join (type a) (x, y : a) (tag : a tag) : a =
  match tag with
  | TInt → x + y
  | TString → x ^ y
```

Full reduction with GADTs

```
type _ tag =
  | TInt : int tag
  | TString : string tag

let join (type a) (x, y : a) (tag : a tag) : a =
  match tag with
  | TInt → x + y                    — we assume a = int
  | TString → x ^ y
```

Full reduction with GADTs

```
type _ tag =
  | TInt : int tag
  | TString : string tag

let join (type a) (x, y : a) (tag : a tag) : a =
  match tag with
  | TInt → x + y
  | TString → x ^ y                — we assume a = string
```

Full reduction with GADTs

```
type _ tag =
  | TInt : int tag
  | TString : string tag

let join (type a) (x, y : a) (tag : a tag) : a =
  match tag with
  | TInt → x + y
  | TString → x ^ y
```

The term (join 3 4) has the following normal form:

```
fun tag →
  match tag with
  | TInt → 3 + 4
  | TString → 3 ^ 4
```

Full reduction with GADTs

```
type _ tag =
  | TInt : int tag
  | TString : string tag

let join (type a) (x, y : a) (tag : a tag) : a =
  match tag with
  | TInt → x + y
  | TString → x ^ y
```

The term (join 3 4) has the following normal form:

```
fun tag →
  match tag with
  | TInt → 3 + 4
  | TString → 3 ^ 4
```

What to do with GADTs?

Should we give up full reduction altogether?

Consider this variant of join:

```
let join (type a) (x, y : a) (tag : a tag) : a * string =
  match tag with
  | TInt    → (x + y, "an" ^ "int")
  | TString → (x ^ y, "a" ^ "string")
```

The string computation does not depend on the assumptions and could be safely reduced.

Our goal

Find the right constructs to allow full reduction in the presence of GADTs.

Implicitly-typed System F with pairs

Terms:

$$a, b \ ::= \ x \mid \lambda(x)\, a \mid a\ a \mid (a, a) \mid \pi_i\, a$$

Evaluation contexts (for full reduction)

$$E \ ::= \ \square \mid \lambda(x)\, E \mid E\ a \mid a\ E \mid (E, a) \mid (a, E) \mid \pi_i\, E$$

Reduction rules:

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

Errors:

$$c \ ::= \ \lambda(x)\, a \mid (a, b) \qquad \text{Constructor expressions}$$

$$D \ ::= \ \square\ a \mid \pi_i\, \square \qquad \text{Destructor contexts}$$

$$\mathcal{E} \ ::= \ \left\{ E\left[\, D[\, c\,]\, \right] \,\middle|\, D[c] \not\circ\!\!\rightarrow \right\} \qquad \text{Errors}$$

Type system

Implicitly typed

$$\tau, \sigma \ ::= \ \alpha \mid \tau \to \sigma \mid \tau * \sigma \mid \forall(\alpha)\, \tau$$

Typing rules

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

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

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

Soundness holds with full reduction

For all variants of System F ($F_{<:}$, MLF, ...)

Type soundness breaks

with inconsistent logical assumptions.

Adding propositions

$$\begin{array}{rll} P & ::= \top \mid P \wedge P \mid \ldots & \text{Logical propositions} \\ & \quad \mid \tau \leq \tau \mid \ldots & \textit{Atomic propositions} \end{array}$$

How can we add support for *logical assumptions* to our system?

Adding propositions

$$\begin{array}{rlr} P & ::= \top \mid P \wedge P \mid \ldots & \text{Logical propositions} \\ & \mid \tau \leq \tau \mid \ldots & \textit{Atomic propositions} \end{array}$$

How can we add support for *logical assumptions* to our system?

$\tau ::= \ldots \mid \forall(\alpha \mid P)\ \tau$ $\quad \Gamma \vdash P$

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

Adding propositions

$$\begin{array}{rcll} P & ::= & \top \mid P \wedge P \mid \ldots & \text{Logical propositions} \\ & \mid & \tau \leq \tau \mid \ldots & \textit{Atomic propositions} \end{array}$$

How can we add support for *logical assumptions* to our system?

$$\tau ::= \ldots \mid \forall(\alpha \mid P)\ \tau \qquad \Gamma \vdash P \qquad \frac{\Gamma \vdash a : \tau \qquad \Gamma \vdash \tau \leq \sigma}{\Gamma \vdash a : \sigma} \qquad \ldots$$

Replacing generalization and instantiation typing rules (the obvious way):

$$\frac{\Gamma, \alpha, P \vdash a : \tau}{\Gamma \vdash a : \forall(\alpha \mid P)\ \tau}\ \textsc{Gen} \qquad \frac{\Gamma \vdash a : \forall(\alpha \mid P)\ \tau \qquad \Gamma \vdash \sigma \qquad \Gamma \vdash P[\sigma/\alpha]}{\Gamma \vdash a : \tau[\sigma/\alpha]}\ \textsc{Inst}$$

Adding propositions

$$\begin{array}{rcll} P & ::= & \top \mid P \wedge P \mid \ldots & \text{Logical propositions} \\ & \mid & \tau \leq \tau \mid \ldots & \textit{Atomic propositions} \end{array}$$

How can we add support for *logical assumptions* to our system?

$$\tau ::= \ldots \mid \boxed{\forall(\alpha \mid P)\ \tau} \qquad \Gamma \vdash P \qquad \frac{\Gamma \vdash a : \tau \qquad \Gamma \vdash \tau \leq \sigma}{\Gamma \vdash a : \sigma} \qquad \ldots$$

Replacing generalization and instantiation typing rules (the obvious way):

$$\frac{\Gamma, \alpha, P \vdash a : \tau}{\Gamma \vdash a : \forall(\alpha \mid P)\ \tau}\ \textsc{Gen} \qquad \frac{\Gamma \vdash a : \forall(\alpha \mid P)\ \tau \qquad \Gamma \vdash \sigma \qquad \boxed{\Gamma \vdash P[\sigma/\alpha]}}{\Gamma \vdash a : \tau[\sigma/\alpha]}\ \textsc{Inst}$$

Subsumes System F, $F_{<:}$, MLF, can encode GADTs:

$$\forall(\alpha \mid \top)\ \sigma \qquad \forall(\alpha \mid \alpha \leq \tau)\ \sigma \qquad \forall(\alpha \mid \alpha \geq \tau)\ \sigma \qquad (\sigma \leq \tau) \wedge (\tau \leq \sigma)$$

The naive rules are unsound

The naive rules are unsound

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

Only *consistent* abstractions are erasable

An abstraction on $(\alpha \mid P)$ is *consistent* when P is satisfied for some type σ substituted for α.

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

Only *consistent* abstractions are erasable

An abstraction on $(\alpha \mid P)$ is *consistent* when P is satisfied for some type σ substituted for α.

$$\frac{\overset{\textsc{Gen}}{}\quad \Gamma, \alpha, P \vdash a : \tau \qquad \boxed{\Gamma \vdash P[\sigma/\alpha]}}{\Gamma \vdash a : \forall(\alpha \mid P)\ \tau}$$

If you cannot prove satisfiability (*e.g.* $\mathbb{B} \leq \mathbb{B} * \mathbb{B}$), you cannot use this rule.

Previous calculi *e.g.* $\mathsf{F}_{<:}$ can still be expressed, since $(\alpha \mid \alpha \leq \sigma)$ is always consistent (satisfied by $\alpha = \sigma$).

Only *consistent* abstractions are erasable

An abstraction on $(\alpha \mid P)$ is *consistent* when P is satisfied for some type σ substituted for α.

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

If you cannot prove satisfiability (*e.g.* $\mathbb{B} \leq \mathbb{B} * \mathbb{B}$), you cannot use this rule.

Previous calculi *e.g.* $\mathsf{F}_{<:}$ can still be expressed, since $(\alpha \mid \alpha \leq \sigma)$ is always consistent (satisfied by $\alpha = \sigma$).

But GADTs cannot be expressed with consistent abstraction only.

Inconsistent abstraction must delay the evaluation.

What is the right design for that?

Only *consistent* abstractions are erasable

An abstraction on $(\alpha \mid P)$ is *consistent* when P is satisfied for some type σ substituted for α.

$$\textsc{Gen}\quad \frac{\Gamma, \alpha, P \vdash a : \tau \qquad \boxed{\Gamma \vdash P[\sigma/\alpha]}}{\Gamma \vdash a : \forall(\alpha \mid P)\ \tau}$$

If you cannot prove satisfiability (*e.g.* $\mathbb{B} \leq \mathbb{B} * \mathbb{B}$), you cannot use this rule.

Previous calculi *e.g.* $\mathsf{F}_{<:}$ can still be expressed, since $(\alpha \mid \alpha \leq \sigma)$ is always consistent (satisfied by $\alpha = \sigma$).

But GADTs cannot be expressed with consistent abstraction only.

Inconsistent abstraction must delay the evaluation.

What is the right design for that?

Explicit vs. Implicit use of hypotheses

In dependently-typed languages, logical propositions are represented as types. Assumptions are introduced using just λ-abstraction $\lambda(z : P)\, a$ and used by explicitly referring to the assumption z:

```
(fun (type a) (x, y : a) (tag : a tag) → match tag with
   | TInt (z : a = int) → (z x) + (z y)
   | TString (z : a = string) → (z x) ^ (z y))
```

If each use of an assumption is marked by a variable,
all dangerous redexes are blocked by those variables.

The same happens in functional intermediate typed representations (*e.g.* System FC).

Explicit vs. Implicit use of hypotheses

In dependently-typed languages, logical propositions are represented as types. Assumptions are introduced using just λ-abstraction $\lambda(z : P)\, a$ and used by explicitly referring to the assumption z:

fun (tag : int tag) $\rightarrow$ **match** tag **with**
  | TInt (z : int = int) $\rightarrow$ (z 3) + (z 4)
  | TString (z : int = string) $\rightarrow$ (z 3) ^ (z 4)

If each use of an assumption is marked by a variable,
all dangerous redexes are blocked by those variables.

The same happens in functional intermediate typed representations (*e.g.* System FC).

Explicit vs. Implicit use of hypotheses

In dependently-typed languages, logical propositions are represented as types. Assumptions are introduced using just λ-abstraction $\lambda(z : P)\, a$ and used by explicitly referring to the assumption z:

fun (tag : int tag) $\rightarrow$ **match** tag **with**
  | TInt (z : int = int) $\rightarrow$ (z 3) + (z 4)
  | TString (z : int = string) $\rightarrow$ (z 3) ^ (z 4)

If each use of an assumption is marked by a variable,
all dangerous redexes are blocked by those variables.

The same happens in functional intermediate typed representations (*e.g.* System FC).

But marking all uses of assumptions explicitly is a burden for the programmer: it is too fine grain.

Explicit vs. Implicit use of hypotheses

In dependently-typed languages, logical propositions are represented as types. Assumptions are introduced using just λ-abstraction $\lambda(z : P)\, a$ and used by explicitly referring to the assumption z:

fun (tag : int tag) $\rightarrow$ **match** tag **with**
  | TInt (z : int = int) $\rightarrow$ (z 3) + (z 4)
  | TString (z : int = string) $\rightarrow$ (z 3) ^ (z 4)

If each use of an assumption is marked by a variable,
all dangerous redexes are blocked by those variables.

The same happens in functional intermediate typed representations (*e.g.* System FC).

But marking all uses of assumptions explicitly is a burden for the programmer: it is too fine grain.

Assumptions should be usable implicitly in derivations, just as consistent abstraction, for both convenience and erasability.

Introducing (possibly) inconsistent assumptions

$$a ::= \ldots \mid \delta(a, \phi.b) \mid \diamond \qquad\qquad \tau ::= \ldots \mid [P]$$

Introducing (possibly) inconsistent assumptions

$$a ::= \ldots \mid \delta(a, \phi.b) \mid \diamond \qquad \tau ::= \ldots \mid [P]$$

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

Introducing (possibly) inconsistent assumptions

$$a ::= \ldots \mid \delta(a, \phi.b) \mid \diamond \qquad \tau ::= \ldots \mid [P]$$

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

Evaluation

$$E ::= \ldots \mid \delta(E, \phi.b) \mid \cancel{\delta(a, \phi.E)} \qquad \delta(\diamond, \phi.b) \hookrightarrow b$$

GADTs, sound edition

```
type 'a tag =
  | TInt of ['a = int]
  | TString of ['a = string]

let join (type a) (x, y : a) (tag : a tag) : a * string =
  match tag with
  | TInt z    → δ(z, ϕ. (x + y, "an" ^ "int"))
  | TString z → δ(z, ϕ. (x ^ y, "a" ^ "string"))
```

We may block the whole branch, as done in OCaml or Haskell

GADTs, sound edition

$$
\begin{array}{l}
\textbf{type}\ \text{'a tag} = \\
\quad \mid \text{TInt}\ \textbf{of}\ [\text{'a} = \text{int}] \\
\quad \mid \text{TString}\ \textbf{of}\ [\text{'a} = \text{string}]
\end{array}
$$

$$
\begin{array}{l}
\textbf{let}\ \text{join}\ (\textbf{type}\ \text{a})\ (\text{x, y : a})\ (\text{tag : a tag}) : \text{a} * \text{string} = \\
\quad \textbf{match}\ \text{tag}\ \textbf{with} \\
\quad \mid \text{TInt z} \quad \rightarrow (\delta(\text{z},\ \phi.\ \text{x} + \text{y}),\ \text{"an"} \mathbin{\hat{}} \text{"int"}) \\
\quad \mid \text{TString z} \rightarrow (\delta(\text{z},\ \phi.\ \text{x} \mathbin{\hat{}} \text{y}),\ \text{"a"} \mathbin{\hat{}} \text{"string"})
\end{array}
$$

We may block the whole branch, as done in OCaml or Haskell

We also offer more flexibility between implicit and explicit use of assumptions.

GADTs, sound edition

```
type 'a tag =
  | TInt of ['a = int]
  | TString of ['a = string]

let join (type a) (x, y : a) (tag : a tag) : a * string =
  match tag with
  | TInt z    → (δ(z, ϕ. x) + δ(z, ϕ. y), "an" ^ "int")
  | TString z → (δ(z, ϕ. x) ^ δ(z, ϕ. y), "a" ^ "string")
```

We may block the whole branch, as done in OCaml or Haskell

We also offer more flexibility between implicit and explicit use of assumptions.

GADTs, sound edition

```
type 'a tag =
  | TInt of ['a = int]
  | TString of ['a = string]

let join (type a) (x, y : a) (tag : a tag) : a * string =
  match tag with
  | TInt z    → δ(z, φ. (x + y, "an" ^ "int"))
  | TString z → δ(z, φ. (x ^ y, "a" ^ "string"))
```

We may block the whole branch, as done in OCaml or Haskell

We also offer more flexibility between implicit and explicit use of assumptions.

Can we do even better?

- leave the use of the assumption implicit in the whole scope
- but say explicitly that "an" ^ "int" is not using the assumption?

Assumption hiding

Assume *only* $\boxed{F}$ is implicitly using the assumption in:

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

Then $\boxed{E}$ and $\boxed{b}$ are unnecessarily blocked.

Assumption hiding

Assume *only* F is implicitly using the assumption in:

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

Then E and b are unnecessarily blocked. We may write

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

Assumption hiding

Assume *only* $\boxed{F}$ is implicitly using the assumption in:

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

Then $\boxed{E}$ and $\boxed{b}$ are unnecessarily blocked. We may write

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

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

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

Assumption hiding

Assume *only* F is implicitly using the assumption in:

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

Then E and b are unnecessarily blocked. We may write

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

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

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

Formally

$$a ::= \ldots \ \mid\ \text{hide}\,\phi\,\text{in}\,b$$

$$\frac{\Gamma \vdash \Delta \qquad \Gamma, \Delta \vdash a : \tau}{\Gamma, \phi : P, \Delta \vdash \text{hide}\,\phi\,\text{in}\,a : \tau}$$

GADTs, last edition

$$
\begin{array}{l}
\textbf{type } \text{'a tag} = \\
\quad \mid \text{TInt } \textbf{of } [\text{'a} = \text{int}] \\
\quad \mid \text{TFloat } \textbf{of } [\text{'a} = \text{float}]
\end{array}
$$

$$
\begin{array}{l}
\textbf{let } \text{join } (\textbf{type } \text{a}) \ (\text{x}, \ \text{y} : \text{a}) \ (\text{tag} : \text{a tag}) : \text{a} * \text{string} = \\
\quad \textbf{match } \text{tag } \textbf{with} \\
\quad \mid \text{TInt z} \quad \rightarrow \delta(\text{z}, \ \phi. \ (\text{x} + \text{y}, \ \text{hide} \, \phi \, \text{in } \text{"an"} \mathbin{\hat{}} \text{"int"})) \\
\quad \mid \text{TString z} \rightarrow \delta(\text{z}, \ \phi. \ (\text{x} \mathbin{\hat{}} \text{y}, \ \text{hide} \, \phi \, \text{in } \text{"a"} \mathbin{\hat{}} \text{"string"}))
\end{array}
$$

The evaluation of "an" ^ "int" need not be blocked anymore.

We offer a continuity between implicit and explicit use of assumptions.

Mixing full and weak reduction: confluence is broken!

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

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

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

Mixing full and weak reduction: confluence is broken!

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

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

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

Idea insert hide ϕ when substitution traverses the guard ϕ.

Mixing full and weak reduction: confluence is restored.

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

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

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

Idea insert hide ϕ when substitution traverses the guard ϕ.

Result The system is *sound* for full-reduction and *confluent*.

Mixing full and weak reduction: confluence is restored.

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

$$\begin{tikzcd}
(\lambda(x)\,\delta(y,\,\phi.\,E[x]))\ a \arrow[r] \arrow[d] & (\lambda(x)\,\delta(y,\,\phi.\,E[x]))\ b \arrow[d] \\
\delta(y,\,\phi.\,E[\mathsf{hide}\,\phi\,\mathsf{in}\ a\ ]) \arrow[r] & \delta(y,\,\phi.\,E\left[\mathsf{hide}\,\phi\,\mathsf{in}\ b\ \right])
\end{tikzcd}$$

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

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

Result The system is *sound* for full-reduction and *confluent*.

Notice: $\mathsf{hide}\,\phi\,\mathsf{in}\,b$ is useful for flexibility, but required for confluence.

Take away

We may support inconsistent abstraction the presence of full reduction.

One must allow to block (for soundness) and *unblock* (for confluence) reduction of subterms.

We should distinguish *consistent* and *inconsistent* abstractions and use whichever is most appropriate.

Language design could help preserve/structure this distinction.

(Opinion) In many cases programmers also think of correctness in an abstract way, in terms of *full reduction*. We should also care for full reduction in the meta-theoretical study of programming languages.

Appendices

We now need to scan under δ's for reducible terms under hides.

$$E ::= \ldots \mid \delta(a,\ \phi.E) \mid \text{hide}\,\phi\,\text{in}\,E$$

In $\delta(a,\ \phi.b)$, b is *guarded* by the assumption variable ϕ, while hide ϕ in a releases the guard ϕ. Evaluation contexts are the unguarded ones:

$$\frac{a \circ\!\!\longrightarrow b \qquad \mathtt{guard}_{\emptyset}(E) = \emptyset}{E[a] \longrightarrow E[b]}\ \textsc{Context}$$

where

$$\begin{array}{lcl}
\mathtt{guard}_S(\lambda(x)\,E) & := & \mathtt{guard}_S(E) \\
\mathtt{guard}_S(\square) & := & S \\
\mathtt{guard}_S(\delta(E,\ \phi.b)) & := & \mathtt{guard}_S(E) \\
\mathtt{guard}_S(\delta(a,\ \phi.E)) & := & \mathtt{guard}_{S,\phi}(E) \\
\mathtt{guard}_S(\text{hide}\,\phi\,\text{in}\,E) & := & \mathtt{guard}_{S \setminus \{\phi\}}(E)
\end{array}$$

We have changed the notion of substitution to insert hidings:

$$(\lambda(x)\, a)\ b \multimap a[b/x]_{\emptyset}$$

$$\begin{array}{lcl}
x[c/y]_S & := & x \\
y[c/y]_S & := & \text{hide}\, S\, \text{in}\, c \\
(\lambda(x)\, a)[c/y]_S & := & \lambda(x)\,(a[c/y]_S) \\
\delta(a,\ \phi.b)[c/y]_S & := & \delta(a[c/y]_S,\ \phi.b[c/y]_{S,\phi}) \\
(\text{hide}\, \phi\, \text{in}\, a)[c/y]_S & := & \text{hide}\, \phi\, \text{in}\, a[c/y]_{S\setminus\{\phi\}}
\end{array}$$

We have also changed the notion of reduction contexts.

These changes are minor as they do not change the term structure, just hiding information, which can be seen as annotations on terms.

Mixing full and weak reduction is a known a problem in the term rewriting community.

In λ-calculus, the solution is to extend weak reduction to allow reduction of subterms under abstractions on which the computation does not depend.

Our solution is somehow similar, but we first had to introduce explicit (blocking and unblocking) marks for logical dependencies.

1. Eliminating hides

We simulate computation with hiding in the language without hiding (and normal β-reduction) by let-extruding hiding constructs.

$$\delta(b,\ \phi.\ E[\text{hide}\,\phi\,\text{in}\ a\ ])$$
$$\hookrightarrow\ \texttt{let}\ x = \quad a \quad \texttt{in}\ \delta(b,\ \phi.\ E[\quad x \quad])$$

If $|a|$ is the $\hookrightarrow$ normal form of a:

- If $a \longrightarrow b$ then $|a| \longrightarrow^* |b|$.
- $a \in \mathit{Errors} \iff |a| \in \mathit{Errors}$

1. Eliminating hides

We simulate computation with hiding in the language without hiding (and normal β-reduction) by let-extruding hiding constructs.

$$\delta(b,\ \phi.\ E[\mathsf{hide}\ \phi\ \mathsf{in}\ a\ ])$$
$$\hookrightarrow\ \texttt{let}\ x = \mathrm{abs}(E, a)\ \texttt{in}\ \delta(b,\ \phi.\ E[\mathrm{app}(x, E)\ ])$$

If $|a|$ is the $\hookrightarrow$ normal form of a:

- If $a \longrightarrow b$ then $|a| \longrightarrow^* |b|$.
- $a \in \mathit{Errors} \iff |a| \in \mathit{Errors}$

2. Soundness of the language without hide

- Bisimulation with Fcc (variant with both consistent and inconsistent abstractions, but no inconsistent assumptions [P])
- Fcc proved sound with a semantics approach.
- Direct soundness proof should be possible.

Consistent assumptions

I focused on (possibly) inconsistent assumptions, but consistent assumptions are also common and equally useful.

Mixing consistent and inconsistent abstraction

We build a data-type α term that contains computations, of type α:

```
type _ term =
  | TLam : 'a * [ 'a = 'b → 'c ] → 'a term
  | TApp : ('b → 'a) term * 'b term → 'a term

let rec eval (type a) (t : a term) : a =
  match t with
  | TLam (f, z) → δ(z, φ. f)
  | TApp (tf, tx) → (eval tf) (eval tx)
```

- The constructor TLam constraints 'a to be an arrow type. A value TLam (f, w) carries a witness z that f has an arrow type.
- The constructor TApp is surjective, so it needs not block the evaluation.

Implicit types

Our calculus is implicitly-typed

- $\odot$ This simplifies the presentation
- $\oplus$ We focus on computation and soundness issues
- $\oplus$ Terms only contain computational constructs (*i.e.* that determines the semantics) and no erasable features at all.
- $\ominus$ Does not provide a surface language.