Full reduction in the face of absurdity

Gabriel Scherer, Didier Rémy

Gallium – INRIA

April 16th, 2015

(Version with notes, for remote reading of the slides.)

"Well-typed programs do not go wrong"

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

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

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

Is this the only point of type systems?

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

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.

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!)

Full reduction in the face of absurdity

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

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!)

Keeping your type system sound for full reduction forces you to detect errors in parts of the program that would not be reduced by a weak reduction strategy. This makes error-checking modular: you are not forced to *use* your functions to see errors in their bodies.

It's hard to convince people with this argument because they've been spoiled by decades of languages sound for full reduction: ML, System F, etc. There's a danger in assuming this just works, yet proving soundness only for call-by-{name,value}. Maybe you let the wolves in without noticing: GADTs break full reduction.

We have other arguments for full reduction. It's a necessary first step equational reasoning for your language. It subsumes soundness proof for call-by-value and call-by-name (while you'll usually only prove soundness for your pet strategy), and it will also tell you whether you soundly support non-deterministic reduction orders. See our paper for more.

```
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
```

Let us study a core language to understand precisely where the unsoundness happens.

$$
\begin{array}{rcll}
a, b & ::= & & \text{Terms} \\
& | & x, y \ldots & \text{variables} \\
& | & \lambda(x)\, a & \lambda\text{-abstraction} \\
& | & a\ b & \text{application} \\
& | & (a, b) & \text{pair} \\
& | & \pi_i\, a & \text{projection}
\end{array}
$$

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

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

$$d ::= \Box\ a \mid \pi_i\, \Box \quad \text{destructor contexts} \qquad c ::= \lambda(x)\, a \mid (a, b) \quad \text{constructors}$$

$$\mathcal{E} \triangleq \{E[d[c]] \mid d[c] \not\circ\!\!\rightarrow\} \quad \text{errors}$$

2017-03-03

Full reduction in the face of absurdity

$$
\begin{array}{rcll}
a, b & ::= & & \text{Terms} \\
& | & x, y \ldots & \text{variables} \\
& | & \lambda(x)\, a & \lambda\text{-abstraction} \\
& | & a\ b & \text{application} \\
& | & (a, b) & \text{pair} \\
& | & \pi_i\, a & \text{projection}
\end{array}
$$

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

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

$d ::= \square\ a \mid \pi_i\, \square$ destructor contexts $\quad c ::= \lambda(x)\, a \mid (a, b)$ constructors

$\mathcal{E} \triangleq \{E[d[c]] \mid d[c] \not\circ\!\!\rightarrow\}$ errors

While usually reduction contexts are used to specify a restricted (often deterministic) reduction strategy, we have every possible term-with-a-hole in our context. That's *full* reduction.

Notice in particular that we allow to reduce under λ (the main point), and reduce non-deterministically on either sides of pairs or applications.

$$
\begin{array}{rcll}
\tau, \sigma & ::= & & \text{Types} \\
& | & \alpha, \beta \ldots & \text{variables} \\
& | & \tau \to \sigma & \text{function types} \\
& | & \tau * \sigma & \text{product types} \\
& | & \forall(\alpha)\ \tau & \text{polymorphism}
\end{array}
$$

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

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

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

Full reduction in the face of absurdity

$$
\begin{array}{rcll}
\tau, \sigma & ::= & & \text{Types} \\
& | & \alpha, \beta \ldots & \text{variables} \\
& | & \tau \to \sigma & \text{function types} \\
& | & \tau * \sigma & \text{product types} \\
& | & \forall(\alpha)\ \tau & \text{polymorphism}
\end{array}
$$

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

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

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

Note that the rule for polymorphic generalization $\Gamma \vdash a : \forall(\alpha)\ \tau$ does *not* change the term typed – this is a Curry-style presentation with no explicit type abstraction in terms. This guarantees by construction that this abstraction is erasable (does not interact with reduction), as reduction is defined only on terms, not type derivations.

All the core calculi you know for programming languages *work fine* with full reduction: simply-typed, ML, System F, $F_{<:}$, MLF ...
It's fun when it *breaks*: adding logical propositions.

$$P, Q \ ::= \ \top \mid P \wedge Q \mid \tau \leq \sigma \mid \ldots \qquad \text{Contexts}$$

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

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

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

Subsumes System F, $F_{<:}$ or GADTs:

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

Problem: this is unsound.

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

We have to restrict these typing rules.

Julien Crétin and Didier Rémy already understood this during Julien's PhD thesis.

An abstraction on $(\alpha \mid P)$ is *consistent* when P is satisfied by some α.
Only *consistent* abstractions are erasable. Others must block reduction.

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

If you cannot prove satisfiability (eg. $\mathbb{B} \leq \mathbb{B} * \mathbb{B}$), you cannot use this rule.
Previous calculi still expressed: $(\alpha \mid \alpha \leq \sigma)$ consistent (satisfied by $\alpha = \sigma$).

But GADTs cannot be expressed with consistent abstraction only.
What is the right design for *inconsistent* abstraction?
This is our new work.

Full reduction in the face of absurdity

Julien Crétin and Didier Rémy already understood this during Julien's PhD thesis.

An abstraction on $(\alpha \mid P)$ is *consistent* when P is satisfied by some α.
Only *consistent* abstractions are erasable. Others must block reduction.

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

If you cannot prove satisfiability (eg. $\mathbb{B} \leq \mathbb{B} * \mathbb{B}$), you cannot use this rule.
Previous calculi still expressed: $(\alpha \mid \alpha \leq \sigma)$ consistent (satisfied by $\alpha = \sigma$).

But GADTs cannot be expressed with consistent abstraction only.
What is the right design for *inconsistent* abstraction?
This is our new work.

Note that not all *inconsistent* abstractions are absurd as in our example. In fact we should think of them as *potentially-inconsistent* abstractions. If you know a proposition will never be true, it makes little sense to abstract on it, it's dead code so the only reasonable thing to do is to return an absurd value with eg. an `assert false` that can be accepted by the type-system when the context is logically absurd.

The interesting cases are when:

- we do not know whether the proposition is satisfiable; eg. making a complexity argument assuming $\mathrm{P} \neq \mathrm{NP}$
- checking satisfiability for the library declarations is too expensive/undecidable, but checking at call site for particular instances is easy
- we want to make an assumption that is unprovable but admissible in the current system, to axiomatize another concept (threads in a programming language with a sequential semantics, excluded middle in a proof system...)

Explicit vs. Implicit

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

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

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.

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

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

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

$$\emptyset \vdash \lambda(x)\, \delta(x, \phi.(\pi_1\, \texttt{true})) : [\mathbb{B} \leq \mathbb{B} * \mathbb{B}] \to \mathbb{B}$$

What you can block, you can un-block

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

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

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

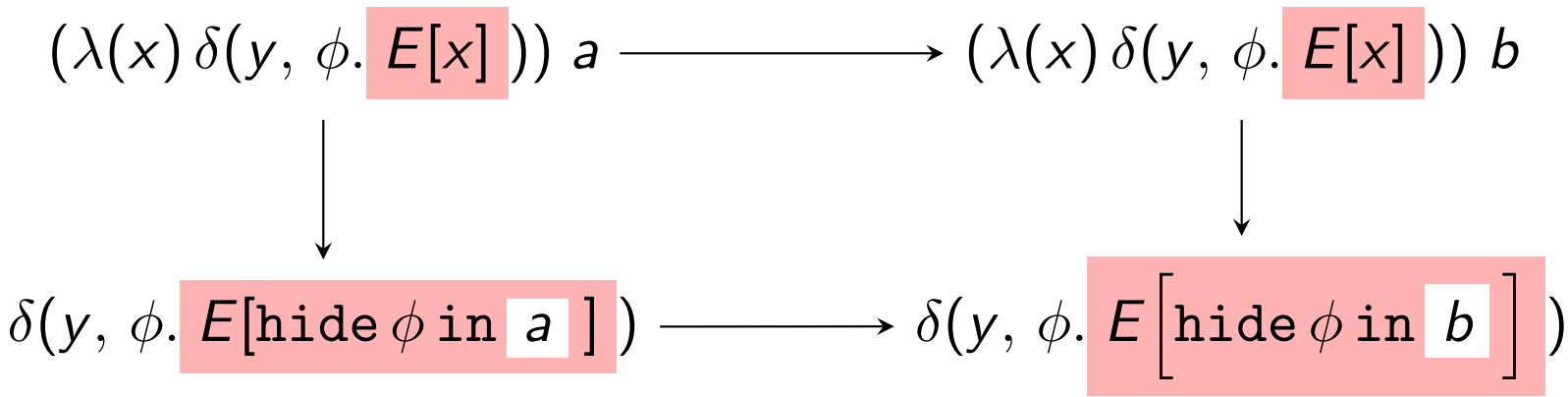
$$E\Big[\delta(a,\ \phi.\ F\Big[\texttt{hide}\,\phi\,\texttt{in}\ b\Big]\ )\Big]$$

$$a +::= \texttt{hide}\,\phi\,\texttt{in}\,b$$

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

Mixing full and weak reduction: confluence in danger!

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



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

Idea: insert $\texttt{hide}\,\phi$ 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)
```

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

Dependable ideas

We call $\Gamma \vdash P$ 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.

Full reduction in the face of absurdity

Dependable ideas

Dependable ideas

We call $\Gamma \vdash P$ 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.

The unability to abstract over definitional equalities has been described a source of difficulty for modular developments in intensional type theories. It could be interesting to study abstraction over consistent equalities. Note that a practical problem with this idea is how to decide the definitional equality under definitional assumptions (eg. by orienting them as rewrite rules); our work does not discuss how to decide the $\Gamma \vdash P$ judgment and is thus orthogonal to this aspect.

In an article near you

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

Integration into an expressive formal framework of *consistent coercion calculi* (Crétin and Rémy): composability of erasable type-system features.

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

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?