Full reduction in the face of absurdity

Gabriel Scherer, Didier Rémy

Gallium – INRIA

October 28, 2014

(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?

$$\lambda(x)\,(\pi_1\ \texttt{true}) \qquad\qquad (\lambda(y)\,2)\ (\lambda(x)\,(\pi_1\ \texttt{true}))$$

Those closed terms cannot evaluate to an error.
Should we improve our type systems to accept them?

$\lambda(x)\,(\pi_1\,\texttt{true})$ $\qquad\qquad$ $(\lambda(y)\,2)\,(\lambda(x)\,(\pi_1\,\texttt{true}))$

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."

Full reduction in the face of absurdity

$\lambda(x)\,(\pi_1\,\texttt{true})$ $\quad$ $(\lambda(y)\,2)\,(\lambda(x)\,(\pi_1\,\texttt{true}))$

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."

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.

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

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

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

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

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

2014-10-28

Full reduction in the face of absurdity

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

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

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

$d ::= \Box\ a \mid \pi_i\, \Box$ 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 generalisation $\Gamma \vdash a : \forall(\alpha)\ \tau$ 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.

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

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: $(\alpha \mid \alpha \leq \sigma)$ always satisfiable (pick $\alpha = \sigma$).

But then, what's 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: $(\alpha \mid \alpha \leq \sigma)$ always satisfiable (pick $\alpha = \sigma$).

But then, what's 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...)

Idea: “box” propositions as values. Unboxing blocks computation.

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

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

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

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

Problem: only doing this breaks confluence.

$$
\begin{array}{lcll}
(\lambda(a)\,\delta(y,\,\phi.a))\;(1+1) & \longrightarrow & (\lambda(a)\,\delta(y,\,\phi.a))\;2 & \longrightarrow \quad \delta(y,\,\phi.2) \\
(\lambda(a)\,\delta(y,\,\phi.a))\;(1+1) & \longrightarrow & \delta(y,\,\phi.(1+1)) & \not\longrightarrow
\end{array}
$$

$(1+1)$ was reducible, but substituting under a δ breaks reducibility.

The solution is to allow to *unblock* reduction of whole subterms by *hiding* some logical assumptions.

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

To define reducibility in this setting, we reason about *guards*: sets of logical assumptions, written S.

$$E +::= \delta(a, \phi.E) \mid \texttt{hide}\,\phi\,\texttt{in}\,E \qquad \frac{a \circ\!\!\rightarrow b \qquad \texttt{guard}_\emptyset(E) = \emptyset}{E[a] \longrightarrow E[b]}$$

$$\begin{aligned}
\texttt{guard}_S(\lambda(x)\,E) &\triangleq \texttt{guard}_S(E) \\
\texttt{guard}_S(\square) &\triangleq S \\
\texttt{guard}_S(\delta(a, \phi.E)) &\triangleq \texttt{guard}_{S \cup \{\phi\}}(E) \\
\texttt{guard}_S(\texttt{hide}\,\phi\,\texttt{in}\,E) &\triangleq \texttt{guard}_{S \setminus \{\phi\}}(E)
\end{aligned}$$

$$\delta(\diamond, \phi.b) \quad \circ\!\!\rightarrow b[\diamond/\phi]$$
$$\texttt{hide}\ \diamond\ \texttt{in}\,a \circ\!\!\rightarrow a$$

Finally, substitution must insert explicit weakenings as necessary – just as De Bruijn shifts.

$$(\lambda(x)\, a)\ b \circ\!\!\rightarrow a[b/x]_\emptyset$$

$$\begin{aligned}
(a\ a')[b/x]_S &\triangleq a[b/x]_S\ a'[b/x]_S \\
(\lambda(y)\, a)[b/x]_S &\triangleq \lambda(y)\, a[b/x]_S
\end{aligned}$$

$$\begin{aligned}
\delta(a,\ \phi.a')[b/x]_S &\triangleq \delta(a[b/x]_S,\ \phi.a'[b/x]_{S\cup\{\phi\}}) \\
(\texttt{hide}\ \phi\ \texttt{in}\ a)[b/x]_S &\triangleq \texttt{hide}\ \phi\ \texttt{in}\ a[b/x]_{S\setminus\{\phi\}}
\end{aligned}$$

$$\begin{aligned}
y[b/x]_S &\triangleq y \\
x[b/x]_S &\triangleq \texttt{hide}\ S\ \texttt{in}\ b
\end{aligned}$$

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

Take away

Studying *full reduction* can tell us new and interesting things about programming languages.

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

To support inconsistent abstraction, one must allow to block *and* unblock reduction of subterms.

Thanks! Questions?