Towards an efficient and formally verified convertibility checker

Nathanaëlle Courant

September 19, 2024

Computation

Let $f : (x, y) \mapsto (x - 1)(y + 1)$.
Let's see how we can compute, for instance, $f(1,\ 27 \cdot 31)$.

Computation

Let $f : (x, y) \mapsto (x - 1)(y + 1)$.
Let's see how we can compute, for instance, $f(1,\ 27 \cdot 31)$.

Bad:

$$f(1,\ 27 \cdot 31) = f(1,\ 837)$$

Computation

Let $f : (x, y) \mapsto (x - 1)(y + 1)$.
Let's see how we can compute, for instance, $f(1,\ 27 \cdot 31)$.

Bad:

$$\begin{aligned} f(1,\ 27 \cdot 31) &= f(1,\ 837) \\ &= (1 - 1)(837 + 1) \end{aligned}$$

Computation

Let $f : (x, y) \mapsto (x - 1)(y + 1)$.
Let's see how we can compute, for instance, $f(1,\ 27 \cdot 31)$.

Bad:

$$\begin{aligned} f(1,\ 27 \cdot 31) &= f(1,\ 837) \\ &= (1 - 1)(837 + 1) \\ &= 0 \cdot (837 + 1) \end{aligned}$$

Computation

Let $f : (x, y) \mapsto (x - 1)(y + 1)$.
Let's see how we can compute, for instance, $f(1,\ 27 \cdot 31)$.

Bad:

$$\begin{aligned} f(1,\ 27 \cdot 31) &= f(1,\ 837) \\ &= (1 - 1)(837 + 1) \\ &= 0 \cdot (837 + 1) \\ &= 0 \end{aligned}$$

Computation

Let $f : (x, y) \mapsto (x - 1)(y + 1)$.
Let's see how we can compute, for instance, $f(1,\ 27 \cdot 31)$.

Bad:

$$\begin{aligned} f(1,\ 27 \cdot 31) &= f(1,\ 837) \\ &= (1 - 1)(837 + 1) \\ &= 0 \cdot (837 + 1) \\ &= 0 \end{aligned}$$

Good:

$$f(1,\ 27 \cdot 31) = (1 - 1)(27 \cdot 31 + 1)$$

Computation

Let $f : (x, y) \mapsto (x - 1)(y + 1)$.
Let's see how we can compute, for instance, $f(1,\ 27 \cdot 31)$.

Bad:

$$\begin{aligned} f(1,\ 27 \cdot 31) &= f(1,\ 837) \\ &= (1 - 1)(837 + 1) \\ &= 0 \cdot (837 + 1) \\ &= 0 \end{aligned}$$

Good:

$$\begin{aligned} f(1,\ 27 \cdot 31) &= (1 - 1)(27 \cdot 31 + 1) \\ &= 0 \cdot (27 \cdot 31 + 1) \end{aligned}$$

Computation

Let $f : (x, y) \mapsto (x - 1)(y + 1)$.
Let's see how we can compute, for instance, $f(1,\ 27 \cdot 31)$.

Bad:

$$\begin{aligned} f(1,\ 27 \cdot 31) &= f(1,\ 837) \\ &= (1 - 1)(837 + 1) \\ &= 0 \cdot (837 + 1) \\ &= 0 \end{aligned}$$

Good:

$$\begin{aligned} f(1,\ 27 \cdot 31) &= (1 - 1)(27 \cdot 31 + 1) \\ &= 0 \cdot (27 \cdot 31 + 1) \\ &= 0 \end{aligned}$$

Computation, a different case

Let $f : x \mapsto x \cdot (100 - x)$.
Let's compute $f(7 \cdot 13 - 1)$.

This time it was better to compute the argument first!

Computation, a different case

Let $f : x \mapsto x \cdot (100 - x)$.
Let's compute $f(7 \cdot 13 - 1)$.

Bad:

$$f(7 \cdot 13 - 1)$$
$$= (7 \cdot 13 - 1)(100 - (7 \cdot 13 - 1))$$

This time it was better to compute the argument first!

Computation, a different case

Let $f : x \mapsto x \cdot (100 - x)$.
Let's compute $f(7 \cdot 13 - 1)$.

Bad:

$$
\begin{aligned}
& f(7 \cdot 13 - 1) \\
&= (7 \cdot 13 - 1)(100 - (7 \cdot 13 - 1)) \\
&= (91 - 1)(100 - (7 \cdot 13 - 1)) \\
&= 90 \cdot (100 - (7 \cdot 13 - 1))
\end{aligned}
$$

This time it was better to compute the argument first!

Computation, a different case

Let $f : x \mapsto x \cdot (100 - x)$.
Let's compute $f(7 \cdot 13 - 1)$.

Bad:

$$
\begin{aligned}
& f(7 \cdot 13 - 1) \\
&= (7 \cdot 13 - 1)(100 - (7 \cdot 13 - 1)) \\
&= (91 - 1)(100 - (7 \cdot 13 - 1)) \\
&= 90 \cdot (100 - (7 \cdot 13 - 1)) \\
&= 90 \cdot (100 - (91 - 1)) \\
&= 90 \cdot (100 - 90)
\end{aligned}
$$

This time it was better to compute the argument first!

Computation, a different case

Let $f : x \mapsto x \cdot (100 - x)$.
Let's compute $f(7 \cdot 13 - 1)$.

Bad:

$$
\begin{aligned}
& f(7 \cdot 13 - 1) \\
&= (7 \cdot 13 - 1)(100 - (7 \cdot 13 - 1)) \\
&= (91 - 1)(100 - (7 \cdot 13 - 1)) \\
&= 90 \cdot (100 - (7 \cdot 13 - 1)) \\
&= 90 \cdot (100 - (91 - 1)) \\
&= 90 \cdot (100 - 90) \\
&= 90 \cdot 10 \\
&= 900
\end{aligned}
$$

This time it was better to compute the argument first!

Computation, a different case

Let $f : x \mapsto x \cdot (100 - x)$.
Let's compute $f(7 \cdot 13 - 1)$.

Bad:

$$
\begin{aligned}
& f(7 \cdot 13 - 1) \\
&= (7 \cdot 13 - 1)(100 - (7 \cdot 13 - 1)) \\
&= (91 - 1)(100 - (7 \cdot 13 - 1)) \\
&= 90 \cdot (100 - (7 \cdot 13 - 1)) \\
&= 90 \cdot (100 - (91 - 1)) \\
&= 90 \cdot (100 - 90) \\
&= 90 \cdot 10 \\
&= 900
\end{aligned}
$$

Good:

$$
\begin{aligned}
& f(7 \cdot 13 - 1) \\
&= f(91 - 1) \\
&= f(90)
\end{aligned}
$$

This time it was better to compute the argument first!

Computation, a different case

Let $f : x \mapsto x \cdot (100 - x)$.
Let's compute $f(7 \cdot 13 - 1)$.

Bad:

$$
\begin{aligned}
& f(7 \cdot 13 - 1) \\
&= (7 \cdot 13 - 1)(100 - (7 \cdot 13 - 1)) \\
&= (91 - 1)(100 - (7 \cdot 13 - 1)) \\
&= 90 \cdot (100 - (7 \cdot 13 - 1)) \\
&= 90 \cdot (100 - (91 - 1)) \\
&= 90 \cdot (100 - 90) \\
&= 90 \cdot 10 \\
&= 900
\end{aligned}
$$

Good:

$$
\begin{aligned}
& f(7 \cdot 13 - 1) \\
&= f(91 - 1) \\
&= f(90) \\
&= 90 \cdot (100 - 90)
\end{aligned}
$$

This time it was better to compute the argument first!

Computation, a different case

Let $f : x \mapsto x \cdot (100 - x)$.
Let's compute $f(7 \cdot 13 - 1)$.

Bad:

$$
\begin{aligned}
& f(7 \cdot 13 - 1) \\
&= (7 \cdot 13 - 1)(100 - (7 \cdot 13 - 1)) \\
&= (91 - 1)(100 - (7 \cdot 13 - 1)) \\
&= 90 \cdot (100 - (7 \cdot 13 - 1)) \\
&= 90 \cdot (100 - (91 - 1)) \\
&= 90 \cdot (100 - 90) \\
&= 90 \cdot 10 \\
&= 900
\end{aligned}
$$

Good:

$$
\begin{aligned}
& f(7 \cdot 13 - 1) \\
&= f(91 - 1) \\
&= f(90) \\
&= 90 \cdot (100 - 90) \\
&= 90 \cdot 10 \\
&= 900
\end{aligned}
$$

This time it was better to compute the argument first!

Equality, up to computation

Let $f : x \mapsto 2^x$.
Let's prove $f(20) = f(19 + 1)$.

First idea: let's compute both sides.

$$f(20) = 2^{20} \qquad f(19+1) = 2^{19+1}$$
$$= 1048576 \qquad = 2^{20}$$
$$= 1048576$$

Costly: we computed 2^{20} on both sides, which was expensive.

Equality, up to computation

Let $f : x \mapsto 2^x$.
Let's prove $f(20) = f(19 + 1)$.

Second idea: if $x = y$, then $f(x) = f(y)$, so we only need to prove $20 = 19 + 1$.

$$20 = 20 \qquad\qquad 19 + 1 = 20$$

That's it, cheap computations only!

Equality, up to computation: the bad case

Let $f : (x, y) \mapsto (x - 1)(y + 1)$.
Let's prove $f(1,\ 27 \cdot 31) = f(1,\ 19 \cdot 47)$.

Let's do as before, and prove $1 = 1$ and $27 \cdot 31 = 19 \cdot 47$.

$$27 \cdot 31 = 837 \qquad\qquad 19 \cdot 47 = 893$$

They are different, so this didn't work!

Equality, up to computation: the bad case

Let $f : (x, y) \mapsto (x - 1)(y + 1)$.
Let's prove $f(1,\ 27 \cdot 31) = f(1,\ 19 \cdot 47)$.

Let's do as before, and prove $1 = 1$ and $27 \cdot 31 = 19 \cdot 47$.

$$27 \cdot 31 = 837 \qquad\qquad 19 \cdot 47 = 893$$

They are different, so this didn't work! We need to unfold f and compute both sides. We already computed both arguments so we don't need to compute them again:

$$\begin{aligned} f(1,\ 837) &= (1 - 1)(837 + 1) \\ &= 0 \cdot (837 + 1) \\ &= 0 \end{aligned} \qquad \begin{aligned} f(1,\ 893) &= (1 - 1)(893 + 1) \\ &= 0 \cdot (893 + 1) \\ &= 0 \end{aligned}$$

We still computed $27 \cdot 31$ and $19 \cdot 47$: expensive.

Equality, up to computation: the bad case

Let $f : (x, y) \mapsto (x - 1)(y + 1)$.
Let's prove $f(1,\ 27 \cdot 31) = f(1,\ 19 \cdot 47)$.

Instead, let's unfold f first.

$$\begin{aligned} f(1,\ 27 \cdot 31) &= (1 - 1)(27 \cdot 31 + 1) \\ &= 0 \cdot (27 \cdot 31 + 1) \\ &= 0 \end{aligned} \qquad \begin{aligned} f(1,\ 19 \cdot 47) &= (1 - 1)(19 \cdot 47 + 1) \\ &= 0 \cdot (19 \cdot 47 + 1) \\ &= 0 \end{aligned}$$

A lot better! But how to know which we should do?
Short answer: we don't, so rely on heuristics (previous works)...
or do it in parallel (this thesis)!

Proof and computation

Suppose we want to prove that $1 + 1 = 2$.

Proof and computation

Suppose we want to prove that $1 + 1 = 2$.

$*54{\cdot}43.\quad \vdash :. \alpha, \beta \in 1 . \supset : \alpha \cap \beta = \Lambda . \equiv . \alpha \cup \beta \in 2$

Dem.

$\vdash . *54{\cdot}26 . \supset \vdash :. \alpha = \iota`x . \beta = \iota`y . \supset : \alpha \cup \beta \in 2 . \equiv . x \neq y .$

$[*51{\cdot}231] \qquad \equiv . \iota`x \cap \iota`y = \Lambda .$

$[*13{\cdot}12] \qquad \equiv . \alpha \cap \beta = \Lambda \qquad (1)$

$\vdash . (1) . *11{\cdot}11{\cdot}35 . \supset$

$\vdash :. (\exists x, y) . \alpha = \iota`x . \beta = \iota`y . \supset : \alpha \cup \beta \in 2 . \equiv . \alpha \cap \beta = \Lambda \qquad (2)$

$\vdash . (2) . *11{\cdot}54 . *52{\cdot}1 . \supset \vdash . \text{Prop}$

From this proposition it will follow, when arithmetical addition has been defined, that $1 + 1 = 2$.

Russel & Whitehead, *Principia Mathematica*, Vol. I (1910), p. 379

Proof and computation

Suppose we want to prove that $1 + 1 = 2$.

More than 300 pages of dense, abstract proofs.
$\Rightarrow$ Can be do better?

Proof and computation

Suppose we want to prove that $1 + 1 = 2$.

More than 300 pages of dense, abstract proofs.
$\Rightarrow$ Can be do better?

Yes! Proof by computation.
Computation: $1 + 1$ is 2, so $1 + 1 = 2$.

Proof and computation

Suppose we want to prove that $1 + 1 = 2$.

> *La « vérification » diffère précisément de la véritable démonstration, parce qu'elle est purement analytique et parce qu'elle est stérile.*
>
> *– H. Poincaré, 1894*

Computation, and equality up to computation, is pure verification: it does not need any ideas.
But it can be can be efficient or not. . .

Proof assistants

What is a proof assistant?

- Tool able to check that proofs (or programs) are correct
- Necessary to specify both statements and proofs; tool checks that the proof is indeed a proof of the statement
- Proofs need to be extremely detailed compared to standard mathematical proofs

Computation is a single proof step $\Longrightarrow$ smaller proofs

Goals for this work

Efficient convertibility test:

- Interactivity of proof assistants: response time is important
- We want worst-case reasonable: time scale $\approx 0.1s$

Trustworthy convertibility test:

- Part of the trusted code base
- Formal verification of the convertibility test is in order

Computation model

What do we need in our computation model?

Computation model

What do we need in our computation model?

- Functions $\lambda x.t$ and function application $f\ x$

Computation model

What do we need in our computation model?

- Functions $\lambda x.t$ and function application $f\ x$
- Defined constants c each with a definition t

Computation model

What do we need in our computation model?

- Functions $\lambda x.t$ and function application $f\ x$
- Defined constants c each with a definition t

For instance: $f \overset{\text{def}}{=} \lambda x.\lambda y.\ \texttt{mul}\ (\texttt{sub}\ x\ 1)\ (\texttt{add}\ y\ 1)$

Computation model

What do we need in our computation model?

- Functions $\lambda x.t$ and function application $f\ x$
- Defined constants c each with a definition t

For instance: $f \overset{\text{def}}{=} \lambda x.\lambda y.\ \texttt{mul}\ (\texttt{sub}\ x\ 1)\ (\texttt{add}\ y\ 1)$

In practice: Coq supports more constructs such as inductive datatypes, so we do too.

Formal statement

λ-calculus with defined constants c, each constant is associated to a term.

Two rules:

- β-reduction: $(\lambda x.t_1)\ t_2 \to_\beta t_1[x := t_2]$
- δ-reduction: if c has definition t, then $c \to_\delta t$

Convertibility test: are two terms $\beta\delta$-equivalent?

Hypothesis: only strongly-normalizing terms $\implies$ decidability.

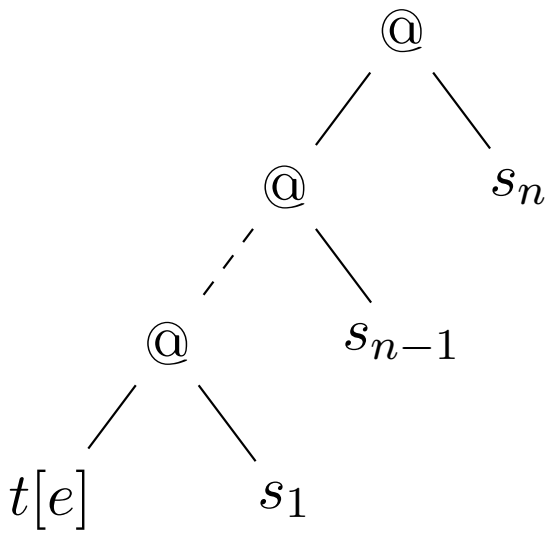
Strong call-by-need

- Efficiency: we want to reduce each subterm at most once, and only if it is needed (call-by-need)
  $\Longrightarrow$ for instance $f(1, 27 \cdot 31)$ as seen before
- Open call-by-need: call-by-need with free variables
  $\Longrightarrow$ for instance, $f(3, x)$ where x is free has normal form: $\texttt{mul}\ 2\ (\texttt{add}\ x\ 1)$
- Strong call-by-need as iterated open call-by-need: we traverse the term, and compute recursively the normal form of each λ-abstraction by applying it to a free variable; need to avoid duplication of work in this step
  $\Longrightarrow$ for instance $\lambda x.f(3, x)$ is in weak but not strong normal form, which is $\lambda x.\texttt{mul}\ 2\ (\texttt{add}\ x\ 1)$

Weak call-by-need evaluation

Evaluation is implemented with a stack in the style of Krivine's machines:

$$\texttt{eval}\ t\ e = \texttt{reduce}\ t\ e\ [\,]$$



Weak call-by-need evaluation

Two operations: building the stack and applying a value to the stack.

$$\begin{aligned}
\texttt{reduce}\ (t\ u)\ e\ s &= \texttt{reduce}\ t\ e\ (\texttt{lazy}\ (\texttt{eval}\ u\ e) :: s) \\
\texttt{reduce}\ (\lambda x.t)\ e\ s &= \texttt{apply}\ \langle x, t, e\rangle\ s \\
\texttt{reduce}\ x\ e\ s &= \texttt{apply}\ (\texttt{force}\ e(x))\ s
\end{aligned}$$

$$\begin{aligned}
\texttt{apply}\ \langle x, t, e\rangle\ (v :: s) &= \texttt{reduce}\ t\ (e + x \mapsto v)\ s \\
\texttt{apply}\ \langle x, t, e\rangle\ [\,] &= \langle x, t, e\rangle
\end{aligned}$$

Encoding in a process calculus

Channels $\alpha, \beta, \ldots$

Processes	$P ::= \bar{\alpha}(M)$	reduce M to a value and send it on α
	$\mid C[\alpha?]$	read a value v from α, continue with $C[v]$
	$\mid P_1 \parallel P_2$	execute P_1 and P_2 concurrently
	$\mid \nu\alpha.P$	generate fresh channel name α

Communication rule:

$$C[\alpha?] \parallel \bar{\alpha}(v) \to C[v] \parallel \bar{\alpha}(v)$$

Channels and processes are used to represent non-strict evaluation:

$$C[\texttt{lazy}\ M] \approx \nu\alpha.\ (C[\alpha] \parallel \bar{\alpha}(M)) \qquad \texttt{force}\ \alpha \approx \alpha?$$

Scheduler enforces laziness: in $\bar{\alpha}(M)$, M does not reduce until α is forced.

Weak call-by-need, process calculus style

$$\bar{\alpha}(\texttt{reduce}\ (t\ u)\ e\ s) \to \nu\beta.\bar{\alpha}(\texttt{reduce}\ t\ e\ (\beta :: s)) \parallel \bar{\beta}(\texttt{reduce}\ u\ e\ [\,])$$

$$\bar{\alpha}(\texttt{reduce}\ (\lambda x.t)\ e\ s) \to \bar{\alpha}(\texttt{apply}\ \langle x, t, e\rangle\ s)$$

$$\bar{\alpha}(\texttt{reduce}\ x\ e\ s) \to \bar{\alpha}(\texttt{apply}\ (e(x))\ s)$$

$$\bar{\alpha}(\texttt{apply}\ \beta\ s) \to \bar{\alpha}(\texttt{apply}\ \beta?\ s)$$

$$\bar{\alpha}(\texttt{apply}\ \langle x, t, e\rangle\ (v :: s)) \to \bar{\alpha}(\texttt{reduce}\ t\ (e + x \mapsto v)\ s)$$

$$\bar{\alpha}(\texttt{apply}\ \langle x, t, e\rangle\ [\,]) \to \bar{\alpha}(\langle x, t, e\rangle)$$

Extending to open call-by-need

New type of (weak) normal forms/values: terms stuck on a free variable in application position, for instance $x\ 1\ 2$

Values: closures $\langle x, t, e\rangle$ or application of a variable to a stack $[x\ s]$

New rules:

$$\bar{\alpha}(\texttt{reduce}\ x\ e\ s) \to \bar{\alpha}(\texttt{apply}\ [x]\ s) \quad \text{if } x \notin \mathbf{dom}(e)$$
$$\bar{\alpha}(\texttt{apply}\ [x\ s_1]\ s_2) \to \bar{\alpha}([x\ (s_1 +\!\!+\ s_2)])$$

Strong call-by-need is iterated open call-by-need

Idea: iterate over the weak normal form, evaluating the body of any function.

Example: $\lambda x.f(3, x)$, already in weak head normal form: generate fresh y, reduce $f(3, y)$ to normal form $\texttt{mul}\ 2\ (\texttt{add}\ y\ 1)$, result is:

$$\lambda y.\ \texttt{mul}\ 2\ (\texttt{add}\ y\ 1)$$

Strong call-by-need is iterated open call-by-need

Idea: iterate over the weak normal form, evaluating the body of any function.

Example: $\lambda x.f(3,x)$, already in weak head normal form: generate fresh y, reduce $f(3,y)$ to normal form $\texttt{mul}\ 2\ (\texttt{add}\ y\ 1)$, result is:

$$\lambda y.\ \texttt{mul}\ 2\ (\texttt{add}\ y\ 1)$$

Problem: need to be careful to avoid loss of sharing! For instance in $(\lambda f.z\,f\,f)\ (\lambda x.t)$, need to avoid duplicating the normalisation of t.

Solution: prepare to compute the normal form of the λ-abstraction when creating the value.

Strong call-by-need: evaluation rules

Function closures are now $\langle x, t, e, y, \beta \rangle$ where β corresponds to the delayed evaluation of $t[x := y]$
$\Longrightarrow \lambda y.\beta?$ corresponds to the normal form of the closure.

Modified rule for evaluation of `reduce`:

$$\bar{\alpha}(\texttt{reduce}\ (\lambda x.t)\ e\ s) \to \nu\beta.\ \bar{\alpha}(\texttt{apply}\ \langle x, t, e, y, \beta \rangle\ s) \parallel \bar{\beta}(\texttt{reduce}\ t\ (e + x \mapsto [y])\ s)$$

Defined constants

Constant c, with definition t.

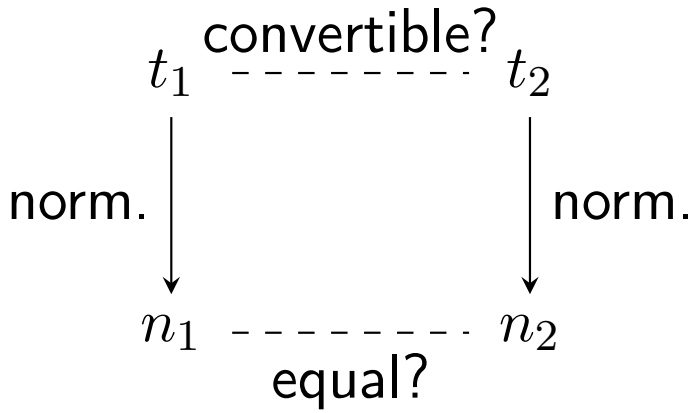
New type of value $[c\ s]@\beta$: constant c applied to stack s.
β is a channel for the evaluation of $t\ s$, after unfolding c.

New rules:

$$\bar{\alpha}(\texttt{reduce}\ c\ e\ s) \to \nu\beta.\ \bar{\alpha}(\texttt{apply}\ [c\ [\,]]@\beta\ s) \parallel \bar{\beta}(\texttt{reduce}\ t\ [\,]\ [\,])$$
$$\bar{\alpha}(\texttt{apply}\ [c\ s_1]@\beta\ s_2) \to \nu\gamma.\ \bar{\alpha}([c\ (s_1 +\!\!+\ s_2)]@\gamma) \parallel \bar{\gamma}(\texttt{apply}\ \beta\ s_2)$$

Testing convertibility, the easy way

We have strong normalisation and confluence: just reduce both sides to normal form and compare the normal forms.



Computing the normal forms can be a lot of work! Sometimes we can conclude non-convertibility after computing the weak head normal forms only. For instance: $x\ t \not\equiv \lambda y.u$ for all t, u.

Idea: we can mix the reduction and the comparison steps! That way, reduction of subterms happens only if necessary.

Testing convertibility, incrementally

Convertibility test $\mathtt{conv}\ v_1\ v_2\ \xi$ between v_1 and v_2, using ξ as a mapping between free variables.

$$\nu\beta\gamma.\ \ \bar{\alpha}(\mathtt{conv}\ \beta\ \gamma\ [\,]) \parallel \bar{\beta}(\mathtt{reduce}\ t_1\ [\,]\ [\,]) \parallel \bar{\gamma}(\mathtt{reduce}\ t_2\ [\,]\ [\,])$$

$\Longrightarrow$ convertibility test between t_1 and t_2, sending the result to α

New rules

For pure λ-calculus:

$$\bar{\alpha}(\texttt{conv}\ \beta\ v_2\ \xi) \to \bar{\alpha}(\texttt{conv}\ \beta?\ v_2\ \xi)$$
$$\bar{\alpha}(\texttt{conv}\ v_1\ \beta\ \xi) \to \bar{\alpha}(\texttt{conv}\ v_1\ \beta?\ \xi)$$

$$\bar{\alpha}(\texttt{conv}\ \langle x_1, t_1, e_1, y_1, v_1\rangle\ \langle x_2, t_2, e_2, y_2, v_2\rangle\ \xi) \to \bar{\alpha}(\texttt{conv}\ v_1\ v_2\ ((y_1, y_2) :: \xi))$$

$$\bar{\alpha}(\texttt{conv}\ [x\ u_1\ \ldots\ u_n]\ [y\ v_1\ \ldots\ v_n]\ \xi) \to \nu\beta_1 \ldots \beta_n.\ \bar{\alpha}\left(\xi(x) = y \wedge \beta_1? \wedge \cdots \wedge \beta_n?\right) \parallel \Vert_{i=1}^{n} \bar{\beta}_i(\texttt{conv}\ u_i\ v_i\ \xi)$$

$$\bar{\alpha}(\texttt{conv}\ v_1\ v_2\ \xi) \to \bar{\alpha}(\texttt{false})$$

all other cases

Short-circuiting operators

We need short-circuiting "and" for early failure when comparing stacks: $\alpha? \wedge \texttt{false} \to \texttt{false}$, without waiting for α

For constants, we will need two other operators $[]$ and $\overrightarrow{\vee}$:

- $A \,[]\, B$ is equal to A but assumes $A = B$, so short-circuits as soon as either A or B is computed.
- $A \overrightarrow{\vee} B$ is $A \vee B$ but assumes $A \Rightarrow B$: short-circuits if A evaluates to $\top$ or if B is computed.

Rules for convertibility of constants

If only one head is a constant, unfold it.
If both heads are constants, we can unfold either and get the same result in any case: use the $[]$ operator to produce a result as soon as possible.

$$\bar{\alpha}(\texttt{conv}\ [c\ s]@\beta\ v_2\ \xi) \to \bar{\alpha}(\texttt{conv}\ \beta\ v_2\ \xi) \text{ if } v_2 \text{ not a constant}$$

$$\bar{\alpha}(\texttt{conv}\ v_1\ [c\ s]@\beta\ \xi) \to \bar{\alpha}(\texttt{conv}\ v_1\ \beta\ \xi) \text{ if } v_1 \text{ not a constant}$$

$$\begin{aligned}\bar{\alpha}(\texttt{conv}\ [c_1\ s_1]@\beta\ [c_2\ s_2]@\gamma\ \xi) \to \nu\delta\eta.\ & \bar{\alpha}(\delta?\ []\ \eta?) \\ \| & \ \bar{\delta}(\texttt{conv}\ [c_1\ s_1]@\beta\ \gamma\ \xi) \\ \| & \ \bar{\eta}(\texttt{conv}\ \beta\ [c_2\ s_2]@\gamma\ \xi) \\ & \text{if } c_1 \neq c_2 \text{ or } |s_1| \neq |s_2|\end{aligned}$$

Rules for convertibility of constants (cont.)

If both heads are the same constant with the same number of arguments, we can also compare the stacks, like with free variables! However terms may be convertible even if the stacks are not, so use $\overrightarrow{\vee}$.

$$\bar{\alpha}(\texttt{conv}\ [c\ u_1\ \ldots\ u_n]@\beta\ [c\ v_1\ \ldots\ v_n]@\gamma\ \xi) \to \nu\delta\eta\zeta_1 \ldots \zeta_n.$$
$$\bar{\alpha}\left(\left(\bigwedge_{i=1}^{n} \zeta_i?\right) \overrightarrow{\vee}(\delta? \,[\!]\, \eta?)\right)$$
$$\| \ \bar{\delta}(\texttt{conv}\ [c\ u_1\ \ldots\ u_n]@\beta\ \gamma\ \xi)$$
$$\| \ \bar{\eta}(\texttt{conv}\ \beta\ [c\ v_1\ \ldots\ v_n]@\gamma\ \xi)$$
$$\| \ \Vert_{i=1}^{n}\ \bar{\zeta_i}(\texttt{conv}\ u_i\ v_i\ \xi)$$

$\Longrightarrow$ exploration of a proof tree!

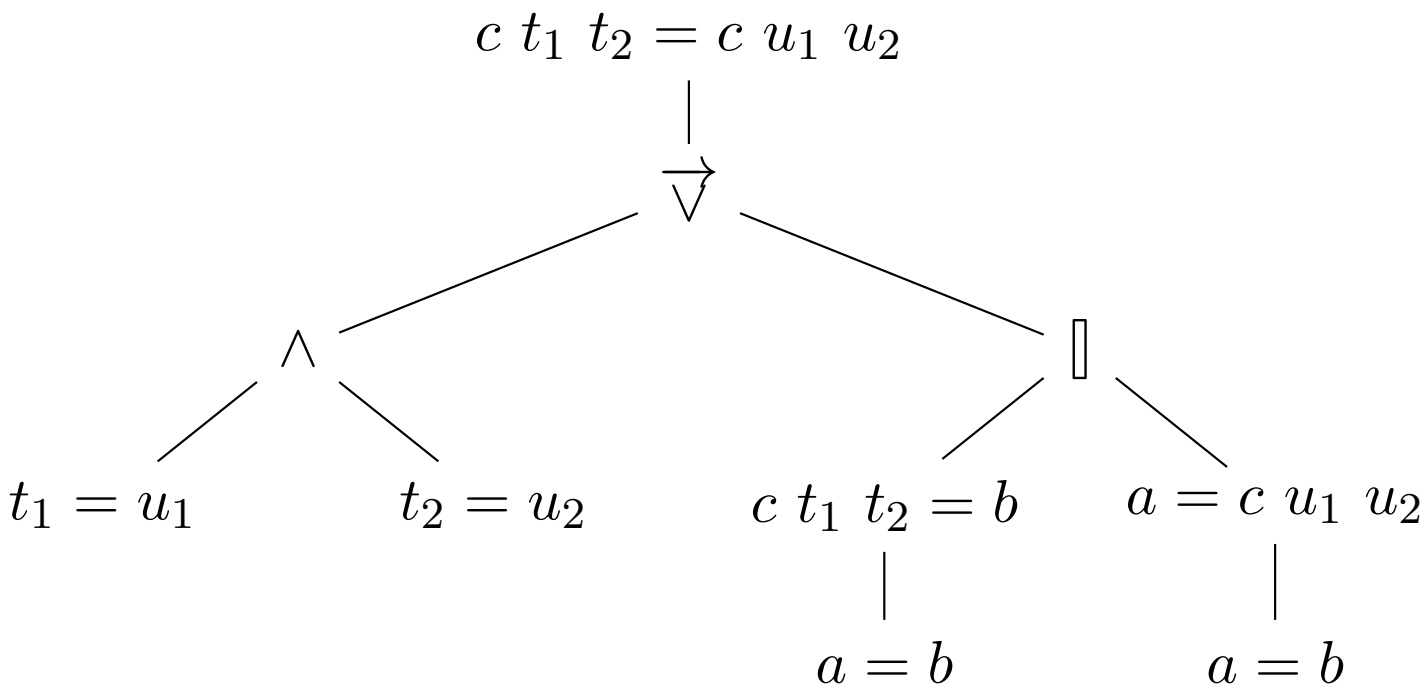
Introduction

Computing normal forms

Testing convertibility

Putting it all together

Proof tree

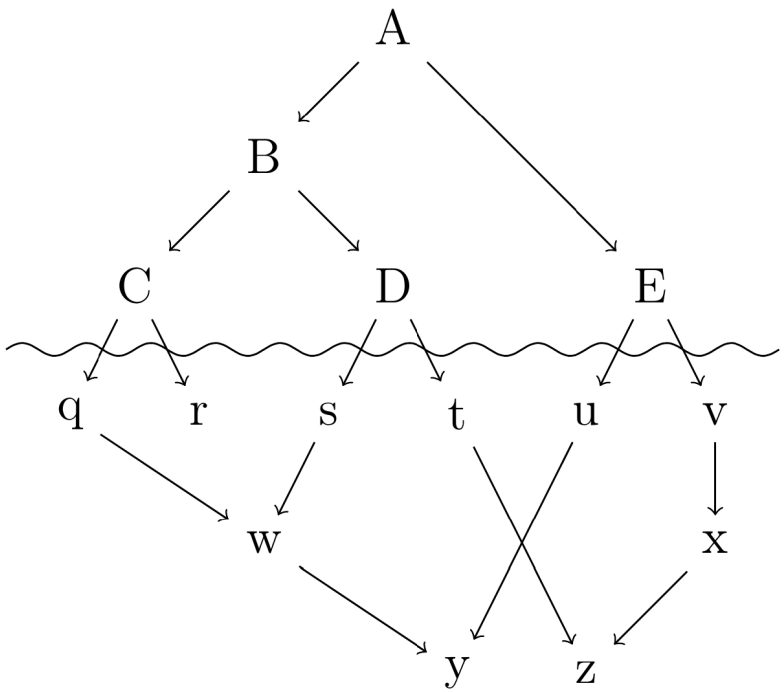

$c\ t_1\ t_2 = c\ u_1\ u_2$
$\overrightarrow{\forall}$
$\wedge$
$\square$
$t_1 = u_1$
$t_2 = u_2$
$c\ t_1\ t_2 = b$
$a = c\ u_1\ u_2$
$a = b$
$a = b$

Sharing of convertibility computations

- With above rules: potential exponential number of convertibility threads due to the unfolding rules
- Limit to only one thread for each pair of computations (v_1, v_2) with a hashconsing-like map
- ξ is not needed in the indexing: always the same mapping for the variables that are free for a given pair (v_1, v_2)!
- Only a $O(n^2)$ convertibility threads for n reduction steps

Thread scheduling

- Lazy values must not be reduced before they are needed
  $\Longrightarrow$ maintain a list of *active* threads
- Simple round-robin scheduling of active threads
  $\Longrightarrow$ ensures fairness
- A thread is *needed* if it is toplevel or there is at least another needed thread waiting for it, *active* if needed and can perform reductions

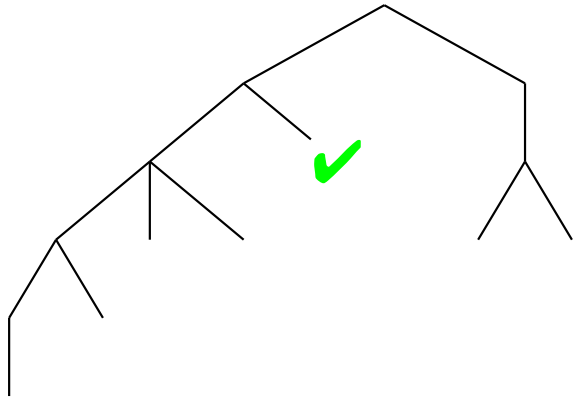


- No cost for lazy values after creation!
- Threads maintain a list of threads to wake up when they finish

Scheduling

- Convertibility rules can remove dependencies on convertibility subthreads
- The hnf of some values might be no longer needed
- Remove convertibility subthreads from the list of threads waiting on a value
- If waiting list becomes empty: remove thread from active list, recursively remove it from waiting list of other threads if applicable
- Thread needs to remain in the current state in case the value is needed again!
- All operations in $O(1)$ with doubly-linked lists

Complexity of convertibility



- If there exists a proof of size n, it will be found at depth at most n
- Branching factor of reduction threads bounded by c
- At depth n: at most $O(c^n)$ reduction threads, $O(c^{2n})$ convertibility threads
- Proof can be found in exponential time of its size (better than Coq)
- In practice: increase in number of threads is small due to sharing

More in the manuscript: Coq formalisation

- Proof of correctness of the reduction machine with respect to β-reduction, and for convertibility
- Extensions to support constructors and (shallow) pattern matching
- No implementation of sharing of convertibility threads or active thread management (non-deterministic semantics)
  $\Longrightarrow$ no proof of complexity
- Proof size: $\approx$ 10k lines

The full statement of the final correctness theorem is thus given below:

```
Lemma all_correct :
  forall defs t1 t2 st b,
    defs_wf defs ->
    closed_at t1 0 -> closed_at t2 0 ->
    dvar_below (length defs) t1 -> dvar_below (length defs) t2 ->
    star step (init_conv defs t1 t2) (cthread_done b, st) ->
    reflect (convertible (betaiota defs) t1 t2) b.
```

It expresses that given two closed terms and well-formed definitions, where the instances of `dvar` inside the terms only reference existing definitions, if after some steps from the initial state created from `defs`, `t1` and `t2` we obtain a `cthread_done b` convertibility

More in the manuscript: a virtual machine

- Faster than interpretation
- Ideas from OCaml and from Coq (`vm_compute`) for further efficiency
- Explicit handling of active threads

	Threads	Active
$\rightsquigarrow_p$	$(r \mapsto \mathtt{R}\ (\mathtt{PUSHRETADDR}(c_2); c_1, e, a, \pi, n, W)) \star T$ $(r \mapsto \mathtt{R}\ (c_1, e, a, \langle c_2, n\rangle :: \pi, n, W)) \star T$	$r :: A$ $A :: r$
$\rightsquigarrow_a$	$(r \mapsto \mathtt{R}\ (\mathtt{LAZY}(c_2); c_1, e, a, \pi, n, W)) \star T$ $(r \mapsto \mathtt{R}\ (c_1, e, a, r_f :: \pi, n, W)) \star (r_f \mapsto \mathtt{R}\ (c_2, e, \emptyset, [], 0, [])) \star T$	$r :: A$ $A :: r$
$\rightsquigarrow_l$	$(r \mapsto \mathtt{R}\ (\mathtt{VAR}(i); c, e, a, \pi, n, W)) \star T$ $(r \mapsto \mathtt{R}\ (c, e, e(i), \pi, n, W)) \star T$	$r :: A$ $A :: r$
$\rightsquigarrow_\lambda$	$(r \mapsto \mathtt{R}\ (\mathtt{ABS}(c_1); c_2, e, a, \pi, n, W)) \star T$ $(r \mapsto \mathtt{R}\ (c_2, e, (c_1, e, y, r_f), \pi, n, W)) \star$ $(r_f \mapsto \mathtt{R}\ (c_1, e, \emptyset, (\mathtt{ACCU}, [], y, \emptyset) :: [], 1, [])) \star T$	$r :: A$ $A :: r$
	$(r \mapsto \mathtt{R}\ (\mathtt{RET}\ c\ y, \langle c, n\rangle :: \pi, 0, W)) \star T$	$r :: A$

Conclusions

Main results:

- New algorithm for testing convertibility of λ-terms with definitions and constructors
- Worst-case bounds on complexity depending on proof size
- Formulated as a machine compatible with compilation
- Formalized and verified in Coq

Take-home messages:

- Complexity of reduction and convertibility matters
- Viewing convertibility algorithms as proof search strategies

Future work

- Extend to other features of Coq: cofixpoints, universes, SProp...
- Implement virtual machine
- Think about native compilation and parallelization
- Improved scheduling of threads
- Producing and replaying proof traces

Experimental evaluation

Testcase	Coq	Ours	Ours2
`test1`	3e-5	5e-5	6e-3
`test1c`	1e-5	5e-5	6e-5
`test2`	0.14	5e-6	2e-5
`test3`	9e-5	2e-4	5e-5
`test4`	0.018	0.013	9e-5
`test5`	4e-6	6e-6	8e-6
`test6`	0.61	1e-6	8e-6
`test7`	3e-5	7e-5	2e-4
`test8`	0.078	5e-5	2e-4
`test9`	2e-5	6e-5/0.18*	0.15

Optimality of call-by-need

Call-by-need: the same redex is reduced at most once, and only if it is needed

What does this mean?

- *same redex* needs to be properly defined: different definitions of optimality depending on this definition
- *only if it is needed*: every reduction path from the input term to a normal form reduces this redex at least once
- in our case: we want to be able to further *compile* terms, so no reductions under a λ-abstraction except to compute its normal form
- Definition of same redex: all λ-abstractions outside the redex have been applied to the same arguments
  - Prevents reducing under the body then applying: two different redexes
  - But allows reducing the body to compute a normal form for strong call-by-need: the λ-abstraction is applied to a free variable when computing the normal form

Most difficult parts of the proof

- For values $\langle x, t, e, y, v \rangle$, where $(\lambda x.t, e)$ is read as t_1 and $\lambda y.v$ as t_2, we only prove $t_1 \equiv t_2$: $t_1 \to^* t_2$ is true but a lot more complicated to prove
- Proving that reducing a thread does not affect the readback of other threads
- Readback has to be specified by a large inductive relation with several nested recursive cases (through `Forall2` for instance), too large to write custom induction principle by hand

Machine reduction rules: base

Threads	Active
$(r \mapsto \mathtt{R}\ (\mathsf{Lazy}(c_2); c_1, e, a, \pi, s, W)) \star T$	$r :: A$
$(r \mapsto \mathtt{R}\ (c_1, e, a, r_f :: \pi, s, W)) \star (r_f \mapsto \mathtt{R}\ (c_2, e, \emptyset, [], [], [])) \star T$	$A :: r$
$(r \mapsto \mathtt{R}\ (\mathsf{Var}(i); c, e, a, \pi, s, W)) \star T$	$r :: A$
$(r \mapsto \mathtt{R}\ (c, e, e(i), \pi, s, W)) \star T$	$A :: r$
$(r \mapsto \mathtt{R}\ (\mathsf{Abs}(c_1); c_2, e, a, \pi, s, W)) \star T$	$r :: A$
$(r \mapsto \mathtt{R}\ (c_2, e, (c_1, e, y, r_f), \pi, s, W)) \star$ $(r_f \mapsto \mathtt{R}\ (c_1, (\mathsf{Accu}, [], y, \emptyset) :: e, \emptyset, [], [], [])) \star T$	$A :: r$
$(r \mapsto \mathtt{R}\ (\mathsf{Ret}, e, v, \pi, c :: s, W)) \star T$	$r :: A$
$(r \mapsto \mathtt{R}\ (c, e, v, \pi, s, W)) \star T$	$A :: r$
$(r \mapsto \mathtt{R}\ (\mathsf{Ret}, e, v, \pi, [], W)) \star T$	$r :: A$
$(r \mapsto \mathtt{D}\ v) \star T$	$A +\!\!+ W$

Machine reduction rules: forcing

Threads	Active
$(r \mapsto \mathtt{R}\ (\mathsf{Force}; c, e, v_{\neg r}, \pi, s, W)) \star T$	$r :: A$
$(r \mapsto \mathtt{R}\ (c, e, v_{\neg r}, \pi, s, W)) \star T$	$A :: r$
$(r \mapsto \mathtt{R}\ (\mathsf{Force}; c, e, r_2, \pi, s, W)) \star (r_2 \mapsto \mathtt{D}\ v) \star T$	$r :: A$
$(r \mapsto \mathtt{R}\ (c, e, v, \pi, s, W)) \star (r_2 \mapsto \mathtt{D}\ v) \star T$	$A :: r$
$(r \mapsto \mathtt{R}\ (\mathsf{Force}; c, e, r_2, \pi, s, W)) \star$ $(r_2 \mapsto \mathtt{R}\ (c, e_2, a_2, \pi_2, s_2, [])) \star T$	$r :: A$
$(r \mapsto \mathtt{R}\ (\mathsf{Force}; c, e, r_2, \pi, s, W)) \star$ $(r_2 \mapsto \mathtt{R}\ (c, e_2, a_2, \pi_2, s_2, r :: [])) \star T$	$A :: r_2$
$(r \mapsto \mathtt{R}\ (\mathsf{Force}; c, e, r_2, \pi, s, W)) \star$ $(r_2 \mapsto \mathtt{R}\ (c, e_2, a_2, \pi_2, s_2, W_2)) \star T$	$r :: A$
$(r \mapsto \mathtt{R}\ (\mathsf{Force}; c, e, r_2, \pi, s, W)) \star$ $(r_2 \mapsto \mathtt{R}\ (c, e_2, a_2, \pi_2, s_2, r :: W_2)) \star T$	A

Machine reduction rules: application

Threads	Active
$(r \mapsto \mathtt{R}\ (\mathsf{App}; c_1, e, (c_2, e_2, y, v_1), v_2 :: \pi, s, W)) \star T$	$r :: A$
$(r \mapsto \mathtt{R}\ (c_2, v_2 :: e_2, (y, v_1), \pi, c_1 :: s, W)) \star T$	$A :: r$
$(r \mapsto \mathtt{R}\ (\mathsf{Accu}, e, (x, \emptyset), \pi, c_1 :: s, W)) \star T$	$r :: A$
$(r \mapsto \mathtt{R}\ (c_1, e, (\mathsf{Accu}, e, x, \emptyset), \pi, s, W)) \star T$	$A :: r$
$(r \mapsto \mathtt{R}\ (\mathsf{AccuConst}, e, (c, v_1), \pi, c_1 :: s, W)) \star T$	$r :: A$
$(r \mapsto \mathtt{R}\ (c_1, e, (\mathsf{AccuConst}, e, c, r_f), \pi, s, W)) \star$ $(r_f \mapsto \mathtt{R}\ (\mathsf{App}; \mathsf{Ret}, e, v_1, e(0) :: [], [], [])) \star T$	$A :: r$