Don't trust, verify: guarantees for the Lean proof assistant

MFoCS Seminar Presentation

Rutger Broekhoff

Radboud University Nijmegen

January 20, 2025

What is Lean?

- A proof assistant
- Heavily used for formalizing mathematics
  - Has an expansive library for mathematics: *mathlib*
- Constructive core, classical community
- CIC-like type theory
  - Impredicative $\mathbb{P}$
  - No universe cumulativity
  - Universe polymorphism
  - Definitional proof irrelevance

A small timeline

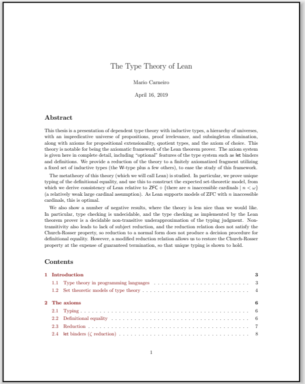
- Started in 2013 by Leonardo de Moura at Microsoft Research
- Lean 0.1 released in 2014
- Lean 2 (0.2) released in 2015
  - Also allowed predicative $\mathbb{P}$
  - Started gathering interest
  - Added support for HoTT
- Lean 3 released in 2017
  - More user-friendly, extensible
  - Removed support for HoTT
  - Separate *mathlib*
- Lean 4 officially released in 2023
  - More like a general-purpose language than previously
  - Extensive macro processor, several kernel extensions
  - Almost completely rewritten in Lean, kernel still in C++

~~Trusting~~ Verifying Lean

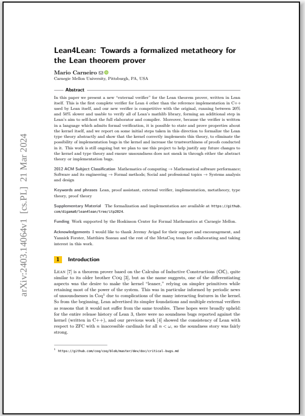
Several stages of enlightenment:

1. A proof assistant written in C++ (2014) ★☆☆☆☆
2. A full mathematical specification of the type theory
3. A consistency proof of the former (both 2019)
4. Only the kernel written in C++ (2023)
5. The kernel also written in Lean (2024) ← We are here!* ★★★⯪☆
6. A mechanized version of the type theory
7. A soundness proof for the kernel
8. A mechanical proof of TT consistency** ★★★★★

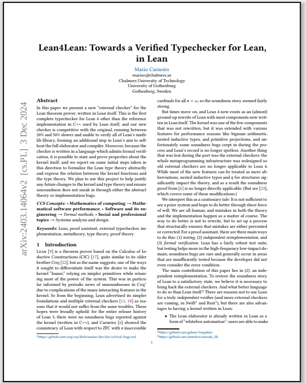
Mario Carneiro's work



Type Theory of Lean: 2019



Lean4Lean (preprint): 2024



Lean4Lean (preprint): 2024

A brief introduction: typing, definitional equality (§§ 2.1, 2.2)

Typing judgement: $\boxed{\Gamma \vdash e : \alpha}$ Definitional equality: $\boxed{\Gamma \vdash e \equiv e'}$

(these are just a select few rules)

CONVERSION

$$\frac{\Gamma \vdash e : \alpha \qquad \Gamma \vdash \alpha \equiv \beta}{\Gamma \vdash e : \beta}$$

TRANSITIVITY

$$\frac{\Gamma \vdash e_1 \equiv e_2 \qquad \Gamma \vdash e_2 \equiv e_3}{\Gamma \vdash e_1 \equiv e_3}$$

β-CONTRACTION

$$\frac{\Gamma, x : \alpha \vdash e : \beta \qquad \Gamma \vdash e' : \alpha}{\Gamma \vdash (\lambda x : \alpha.\ e)\ e' \equiv e[e'/x]}$$

PROOF IRRELEVANCE

$$\frac{\Gamma \vdash p : \mathbb{P} \qquad \Gamma \vdash h : p \qquad \Gamma \vdash h' : p}{\Gamma \vdash h \equiv h'}$$

Proof irrelevance

Say we define $\Sigma x : \alpha.\ B(x)$ as follows:

$$\alpha : \mathsf{U}_{\ell_1}, \beta : \alpha \to \mathsf{U}_{\ell_2} \vdash \mathsf{sig}_{\alpha\beta} := \mu t : \mathsf{U}_{\mathsf{max}(\ell_1,\ell_2,1)}.\ (\mathsf{exist} : \forall x : \alpha.\ \beta\ x \to t)$$

- Now, say we let $\alpha := \mathbb{N}, \beta := \lambda x.\ x < 10$.
- Then, *e.g.*, $\mathsf{exist}_{\mathsf{sig}}\ 5\ h_1 \equiv \mathsf{exist}_{\mathsf{sig}}\ 5\ h_2$ by compatibility and proof irrelevance.

A brief introduction: algorithmic equality (§ 2.3)

Definitional equality: $\boxed{\Gamma \vdash e \equiv e'}$ vs. Algorithmic equality: $\boxed{\Gamma \vdash e \Leftrightarrow e'}$

(these are just a select few rules)

$$\frac{\Gamma \vdash e_1 \equiv e_2 \qquad \Gamma \vdash e_2 \equiv e_3}{\Gamma \vdash e_1 \equiv e_3}\ \text{TRANSITIVITY}$$

$\Leftrightarrow$ lacks an explicit transitivity rule

$$\frac{\Gamma, x : \alpha \vdash e : \beta \qquad \Gamma \vdash e' : \alpha}{\Gamma \vdash (\lambda x : \alpha.\ e)\ e' \equiv e[e'/x]}\ \beta\text{-CONTRACTION}$$

$$\frac{e \rightsquigarrow k \qquad \Gamma \vdash k \Leftrightarrow e'}{\Gamma \vdash e \Leftrightarrow e'}\ \text{REDUCTION}$$

Properties of Lean's type theory

- Definitional equality is undecidable
- Algorithmic equality is not transitive, *i.e.*,

$$\exists e_1, e_2, e_3.\ e_1 \Leftrightarrow e_2 \to e_2 \Leftrightarrow e_3 \to e_1 \not\Leftrightarrow e_3$$

- Subject reduction fails in practice, *i.e.*,

$$\exists e, e'.\ (\Gamma \Vdash e : \alpha) \to e \leadsto e' \to \Gamma \nVdash e' : \alpha$$

- Normalization fails, *i.e.*,

$$\exists e.\ \neg\exists e'.\ e \equiv e' \wedge e' \not\leadsto_{\beta\delta\iota(\zeta\eta)}$$

Time for examples: undecidability of $\equiv$

Three ingredients:

- Something that can unfold forever in an inconsistent context: *accessibility on a relation that is not well-founded* ($>$ on $\mathbb{N}$)
- To control unfolding: a predicate $P : \mathbb{N} \to \mathbf{2}$ that is decidable, but for which $\forall n : \mathbb{N}, P\ n$ is undecidable: "*Turing machine M runs for at least n steps without halting*"
- A function that combines both

Time for examples: undecidability of $\equiv$

Three ingredients:

- **Something that can unfold forever in an inconsistent context:** ***accessibility on a relation that is not well-founded*** **($>$ on $\mathbb{N}$)**
- To control unfolding: a predicate $P : \mathbb{N} \to \mathbf{2}$ that is decidable, but for which $\forall n : \mathbb{N}, P\ n$ is undecidable: "*Turing machine M runs for at least n steps without halting*"
- A function that combines both

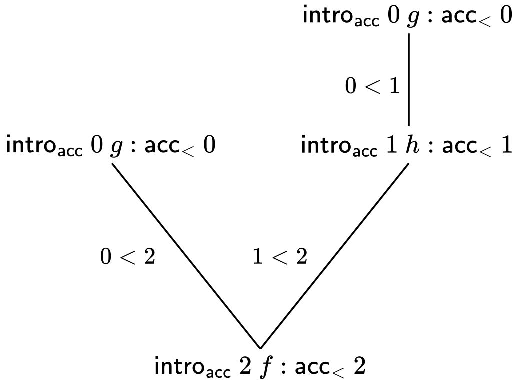
Accessibility with $<$ (WF)

$$\mathsf{acc}_< := \mu A : \mathbb{N} \to \mathbb{P}.\ (\mathsf{intro} : \forall x : \mathbb{N}.\ (\forall y : \mathbb{N}.\ y < x \to A\ y) \to A\ x)$$

$$\begin{aligned}\mathsf{rec}_{\mathsf{acc}} : \forall C : \mathbb{N} \to \mathsf{U}_u.\ (\forall x : \mathbb{N}.\ & (\forall y : \mathbb{N}. y < x \to \mathsf{acc}_<\ y) \to \\ & (\forall y : \mathbb{N}. y < x \to C\ y) \to \\ & C\ x) \to \\ \forall n : \mathbb{N}.\ & \mathsf{acc}_<\ n \to C\ n\end{aligned}$$

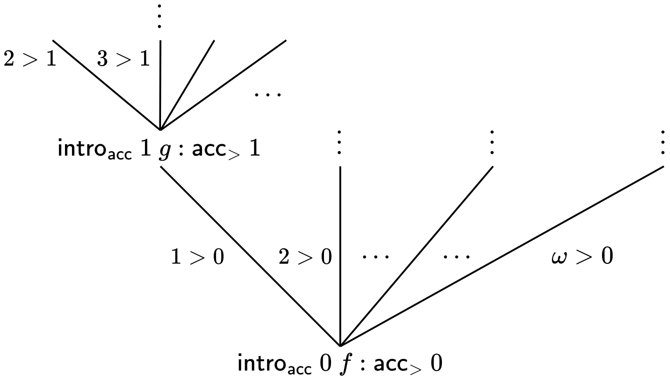
Accessibility with < (WF)

$$acc_< := \mu A : \mathbb{N} \to \mathbb{P}.\ (\mathsf{intro} : \forall x : \mathbb{N}.\ (\forall y : \mathbb{N}.\ y < x \to A\ y) \to A\ x)$$



Accessibility with > (not WF)

$$\mathsf{acc}_{>} := \mu A : \mathbb{N} \to \mathbb{P}.\ (\mathsf{intro} : \forall x : \mathbb{N}.\ (\forall y : \mathbb{N}.\ y > x \to A\ y) \to A\ x)$$



Inversion on accessibility

- We can project out the second argument to intro using inv_x:

$$\mathsf{inv}_x : \mathsf{acc}\ x \to \forall y : \mathbb{N}.\ y < x \to \mathsf{acc}\ y$$

  (inv_x is defined using $\mathsf{rec}_{\mathsf{acc}}$, but its exact definition is not that important)
- Additionally:

$$a \equiv \mathsf{intro}_{\mathsf{acc}}\ x\ (\mathsf{inv}_x\ a) : \mathsf{acc}\ x : \mathbb{P}$$

$$\textsc{Proof Irrelevance}\quad \frac{\Gamma \vdash p : \mathbb{P} \qquad \Gamma \vdash h : p \qquad \Gamma \vdash h' : p}{\Gamma \vdash h \equiv h'}$$

Time for examples: undecidability of $\equiv$

Three ingredients:

- Something that can unfold forever in an inconsistent context: *accessibility on a relation that is not well-founded* ($>$ on $\mathbb{N}$)
- To control unfolding: a predicate $P : \mathbb{N} \to \mathbf{2}$ that is decidable, but for which $\forall n : \mathbb{N}, P\ n$ is undecidable: "*Turing machine M runs for at least n steps without halting*"
- A function that combines both

Time for examples: undecidability of $\equiv$

Three ingredients:

- Something that can unfold forever in an inconsistent context:
  accessibility on a relation that is not well-founded ($>$ on $\mathbb{N}$) ✓
- To control unfolding: a predicate $P : \mathbb{N} \to \mathbf{2}$ that is decidable,
  but for which $\forall n : \mathbb{N}, P\ n$ is undecidable:
  "Turing machine M runs for at least n steps without halting"
- A function that combines both

Time for examples: undecidability of $\equiv$

Three ingredients:

- Something that can unfold forever in an inconsistent context:
  accessibility on a relation that is not well-founded ($>$ on $\mathbb{N}$) ✓
- To control unfolding: a predicate $P : \mathbb{N} \to \mathbf{2}$ that is decidable,
  but for which $\forall n : \mathbb{N}, P\ n$ is undecidable:
  "*Turing machine M runs for at least n steps without halting*" ✓
- A function that combines both

Time for examples: undecidability of $\equiv$

Three ingredients:

- Something that can unfold forever in an inconsistent context: *accessibility on a relation that is not well-founded* ($>$ on $\mathbb{N}$) ✓
- To control unfolding: a predicate $P : \mathbb{N} \to \mathbf{2}$ that is decidable, but for which $\forall n : \mathbb{N}, P\ n$ is undecidable: "*Turing machine M runs for at least n steps without halting*" ✓
- **A function that combines both**

Time for examples: undecidability of $\equiv$

$$
\begin{aligned}
&f : \forall n.\ \mathsf{acc}_{>}\ n \to \mathbf{1} \\
&f := \mathsf{rec}_{\mathsf{acc}}\ (\lambda\_.\ \mathbf{1})\ (\lambda n\,\_\,(g : \forall y.\ y > x \to \mathbf{1}). \\
&\qquad\qquad \text{if } P\ n \text{ then } g\ (n+1)\ (p\ n) \text{ else } ())
\end{aligned}
$$

Time for examples: undecidability of $\equiv$

$$\begin{aligned} f &: \forall n.\ \mathsf{acc}_{>}\ n \to \mathbf{1} \\ f &:= \mathsf{rec}_{\mathsf{acc}}\ (\lambda\_.\ \mathbf{1})\ (\lambda n\ \_\ (g : \forall y.\ y > x \to \mathbf{1}). \\ &\qquad \text{if } P\ n \text{ then } g\ (n+1)\ (p\ n) \text{ else } ()) \end{aligned}$$

$$f\ n\ (\mathsf{intro}_{\mathsf{acc}}\ n\ h) \rightsquigarrow^{*} \text{if } P\ n \text{ then } f\ (n+1)\ (h\ (n+1)\ (p\ n)) \text{ else } ()$$

Time for examples: undecidability of $\equiv$

(recall $a \equiv \mathsf{intro}_{\mathsf{acc}}\ x\ (\mathsf{inv}_x\ a)$)

$$
\begin{aligned}
f &: \forall n.\ \mathsf{acc}_{>}\ n \to \mathbf{1} \\
f &:= \mathsf{rec}_{\mathsf{acc}}\ (\lambda\_.\ \mathbf{1})\ (\lambda n\_(g : \forall y.\ y > x \to \mathbf{1}). \\
&\qquad \text{if } P\ n \text{ then } g\ (n+1)\ (p\ n) \text{ else } ())
\end{aligned}
$$

$$
f\ n\ (\mathsf{intro}_{\mathsf{acc}}\ n\ h) \rightsquigarrow^{*} \text{if } P\ n \text{ then } f\ (n+1)\ (h\ (n+1)\ (p\ n)) \text{ else } ()
$$

$$
\begin{aligned}
f\ 0\ a &\equiv f\ 0\ (\mathsf{intro}_{\mathsf{acc}}\ 0\ (\mathsf{inv}_0\ a)) \\
&\equiv f\ 1\ (\mathsf{inv}_0\ a\ 1\ (p\ 0)) \\
&\equiv f\ 1\ (\mathsf{intro}_{\mathsf{acc}}\ 1\ (\mathsf{inv}_1\ (\mathsf{inv}_0\ a\ 1\ (p\ 0))) \\
&\equiv f\ 2\ (\mathsf{inv}_1\ (\mathsf{inv}_0\ a\ 1\ (p\ 0))\ 2\ (p\ 1)) \\
&\equiv \ldots
\end{aligned}
$$

Time for examples: undecidability of $\equiv$

(recall $a \equiv \text{intro}_{\text{acc}}\ x\ (\text{inv}_x\ a)$)

$$f : \forall n.\ \text{acc}_{>}\ n \to \mathbf{1}$$

$$f := \text{rec}_{\text{acc}}\ (\lambda\_.\ \mathbf{1})\ (\lambda n\,\_\,(g : \forall y.\ y > x \to \mathbf{1}).\ \text{if } P\ n \text{ then } g\ (n+1)\ (p\ n) \text{ else } ())$$

$$f\ n\ (\text{intro}_{\text{acc}}\ n\ h) \rightsquigarrow^{*} \text{if } P\ n \text{ then } f\ (n+1)\ (h\ (n+1)\ (p\ n)) \text{ else } ()$$

$$\begin{aligned} f\ 0\ a &\equiv f\ 0\ (\text{intro}_{\text{acc}}\ 0\ (\text{inv}_0\ a)) \\ &\equiv f\ 1\ (\text{inv}_0\ a\ 1\ (p\ 0)) \\ &\equiv f\ 1\ (\text{intro}_{\text{acc}}\ 1\ (\text{inv}_1\ (\text{inv}_0\ a\ 1\ (p\ 0)))) \\ &\equiv f\ 2\ (\text{inv}_1\ (\text{inv}_0\ a\ 1\ (p\ 0))\ 2\ (p\ 1)) \\ &\equiv \dots \end{aligned}$$

$$a : \text{acc}_{>}\ 0 \vdash f\ 0\ a \equiv () \quad \text{if and only if} \quad \neg\forall n.\ P\ n$$

Algorithmic equality is not transitive (1)

$$\begin{aligned} f\ 0\ a &\equiv f\ 0\ (\mathsf{intro}_{\mathsf{acc}}\ 0\ (\mathsf{inv}_0\ a)) \\ &\equiv f\ 1\ (\mathsf{inv}_0\ a\ 1\ (p\ 0)) \\ &\equiv f\ 1\ (\mathsf{intro}_{\mathsf{acc}}\ 1\ (\mathsf{inv}_1\ (\mathsf{inv}_0\ a\ 1\ (p\ 0)))) \\ &\equiv f\ 2\ (\mathsf{inv}_1\ (\mathsf{inv}_0\ a\ 1\ (p\ 0))\ 2\ (p\ 1)) \\ &\equiv \dots \end{aligned}$$

$$\begin{aligned} f\ 0\ a &\Leftrightarrow f\ 0\ (\mathsf{intro}_{\mathsf{acc}}\ 0\ (\mathsf{inv}_0\ a)) \\ &\Leftrightarrow f\ 1\ (\mathsf{inv}_0\ a\ 1\ (p\ 0)) \end{aligned}$$

but

$$f\ 0\ a \not\Leftrightarrow f\ 1\ (\mathsf{inv}_0\ a\ 1\ (p\ 0))$$

$$\begin{aligned} f &: \forall n.\ \mathsf{acc}_{>}\ n \to \mathbf{2} \\ f &:= \mathsf{rec}_{\mathsf{acc}}\ (\lambda\_.\ \mathbf{2})\ (\lambda n\, \_\, (g : \forall y.\ y > x \to \mathbf{2}). \\ &\qquad \mathsf{if}\ P\ n\ \mathsf{then}\ g\ (n+1)\ (p\ n)\ \mathsf{else}\ \mathsf{tt}) \end{aligned}$$

$$f\ n\ (\mathsf{intro}_{\mathsf{acc}}\ n\ h) \rightsquigarrow^{*} \mathsf{if}\ P\ n\ \mathsf{then}\ f\ (n+1)\ (h\ (n+1)\ (p\ n))\ \mathsf{else}\ \mathsf{tt}$$

Algorithmic equality is not transitive (2)

Also in a consistent context ($a : \mathrm{acc}_< \ 1$):

$$
\begin{aligned}
f\ 1\ a &\Leftrightarrow f\ 1\ (\mathrm{intro}_{\mathrm{acc}}\ 1\ (\mathrm{inv}_1\ \ a)) \\
&\Leftrightarrow f\ 0\ (\mathrm{inv}_1\ \ a\ 0\ p'_0) \\
&\text{but} \\
f\ 1\ a &\not\Leftrightarrow f\ 0\ (\mathrm{inv}_1\ \ a\ 0\ p'_0)
\end{aligned}
$$

$$
\begin{aligned}
&f : \forall n.\ \mathrm{acc}_<\ \ n \rightarrow \mathbf{2} \\
&f := \mathrm{rec}_{\mathrm{acc}}\ (\lambda\_.\ \mathbf{2})\ (\lambda n\,\_(g : \forall y.\ y < x \rightarrow \mathbf{2}). \\
&\qquad\qquad \text{if } P\ \ n \text{ then if } h : n \neq 0 \text{ then } g\ (n-1)\ (p'\ \ n\ \ h) \text{ else tt else tt})
\end{aligned}
$$

$$
f\ \ n\,(\mathrm{intro}_{\mathrm{acc}}\ n\ h) \rightsquigarrow^* \text{if } P\ n \text{ then if } h : n \neq 0 \text{ then } f\ (n{-}1)\ (h\ (n{-}1)\ (p\ n\ h)) \text{ else tt else tt}
$$

Subject reduction fails in practice (1)

- So far, we have demonstrated lack of transitivity for inhabitants of $\mathbb{B} : \mathsf{U}_1$.
- We can use the same strategy to synthesize 'problematic' types in a consistent context:

$$\varphi : \forall n.\ \mathsf{acc}_< \ n \to \mathsf{U}_1$$
$$\varphi := \mathsf{rec}_{\mathsf{acc}}\ (\lambda\_.\ \mathsf{U}_1)\ (\lambda n\,\_\,(g : \forall y.\ y < x \to \mathsf{U}_1).$$
$$\text{if } P\ n \text{ then if } h : n \neq 0 \text{ then } g\ (n-1)\ (p'\ n\ h) \text{ else } \mathbb{N} \text{ else } \mathbb{N})$$

- We know that $\|\mathsf{acc}_< \ 1\|$, so assume we have $a : \mathsf{acc}_< \ 1$ (a may not be transparent).
- Then define:

$$\left.\begin{aligned} \alpha &:= \varphi\ 1\ a \\ \beta &:= \varphi\ 1\ (\mathsf{intro}_{\mathsf{acc}}\ 1\ (\mathsf{inv}_1\ a)) \\ \gamma &:= \varphi\ 0\ (\mathsf{inv}_1\ a\ 0\ p_0') \end{aligned}\right\} \quad \text{so we have} \quad \begin{cases} \alpha \Leftrightarrow \beta \\ \quad \Leftrightarrow \gamma \\ \alpha \not\Leftrightarrow \gamma \end{cases}$$

Subject reduction fails in practice (2)

- Let $\Gamma \Vdash e : \alpha$ denote the algorithmic typing judgement that Lean uses, like the normal typing judgement, but using $\Leftrightarrow$ instead of $\equiv$ in the conversion rule.
- Recall $\Gamma \vdash \alpha \Leftrightarrow \beta; \beta \Leftrightarrow \gamma; \alpha \not\Leftrightarrow \gamma$.
- We can prove that $\alpha, \beta, \gamma = \mathbb{N}$, so we can cast, *e.g.*, $0 : \mathbb{N}$ to $0 : \gamma$. Assume $\Gamma \Vdash e : \gamma$.
- Now:

  $\Gamma \Vdash \mathrm{id}_\beta\ e : \beta$, checks $\Gamma \vdash \beta \Leftrightarrow \gamma$
  $\Gamma \Vdash \mathrm{id}_\alpha\ (\mathrm{id}_\beta\ e) : \alpha$, checks $\Gamma \vdash \alpha \Leftrightarrow \beta$
  $\Gamma \nVdash \mathrm{id}_\alpha\ e : \alpha$, checks $\Gamma \vdash \alpha \Leftrightarrow \gamma$ but $\Gamma \vdash \alpha \not\Leftrightarrow \gamma$
- But $\mathrm{id}_\alpha\ (\mathrm{id}_\beta\ e) \Leftrightarrow \mathrm{id}_\alpha\ e$ since $\mathrm{id}_\beta\ e \rightsquigarrow e$.
- So $\Gamma \Vdash \mathrm{id}_\alpha\ (\mathrm{id}_\beta\ e) : \alpha$ and $\mathrm{id}_\alpha\ (\mathrm{id}_\beta\ e) \Leftrightarrow \mathrm{id}_\alpha\ e$, but not $\Gamma \Vdash \mathrm{id}_\alpha\ e : \alpha$.

Subject reduction does not fail in theory

- As demonstrated here:

$$\dfrac{\dfrac{\Gamma \Vdash \mathsf{id}_\alpha : \forall x : \alpha.\ \alpha \qquad \dfrac{\Gamma \vdash \alpha \Leftrightarrow \beta \qquad \Gamma, x : \alpha \vdash \alpha \Leftrightarrow \alpha}{\Gamma \vdash \forall x : \alpha.\ \alpha \Leftrightarrow \forall x : \beta.\ \alpha}}{\Gamma \Vdash \mathsf{id}_\alpha : \forall x : \beta.\ \alpha} \qquad \dfrac{\Gamma \Vdash e : \gamma \qquad \Gamma \vdash \beta \Leftrightarrow \gamma}{\Gamma \Vdash e : \beta}}{\Gamma \Vdash \mathsf{id}_\alpha\ e : \alpha}$$

Subject reduction does not fail in theory

- As demonstrated here:

$$\dfrac{\Gamma \Vdash \mathrm{id}_\alpha : \forall x : \alpha.\ \alpha \qquad \dfrac{\Gamma \vdash \alpha \Leftrightarrow \beta \qquad \Gamma, x : \alpha \vdash \alpha \Leftrightarrow \alpha}{\Gamma \vdash \forall x : \alpha.\ \alpha \Leftrightarrow \forall x : \beta.\ \alpha}}{\Gamma \Vdash \mathrm{id}_\alpha : \forall x : \beta.\ \alpha} \qquad \dfrac{\Gamma \Vdash e : \gamma \qquad \Gamma \vdash \beta \Leftrightarrow \gamma}{\Gamma \Vdash e : \beta}$$
$$\overline{\Gamma \Vdash \mathrm{id}_\alpha\ e : \alpha}$$

- Conclusion: Carneiro's algorithmic type judgement is less strict than the one Lean uses internally.
- This is totally fine as long as we consider an overapproximation of what Lean does.
  - (This is actually not the case: Lean considers $a, b : \mathbf{1} : \mathrm{U}_1 \Vdash a \equiv b$, but this may have not been the case in Lean 3.)

Abel & Coquand nontermination proof

- Type theories with proof irrelevance and impredicative $\mathbb{P}$ lose strong normalization.
  - Lean falls into this category.
  - So does Coq with SProp, as long as you enable definitional UIP.
- Two variants:
  1. using the absurdity that all propositions are equal, and
  2. using (weak) propositional extensionality.

Abel & Coquand nontermination proof: exhibit A

```
Set Definitional UIP.

Inductive seq {A} (a : A) : A → SProp :=
  srefl : seq a a.

Definition cast (A B : Prop) (e : seq A B) (x : A) : B :=
  match e with srefl _ ⇒ x end.

Definition False : Prop := ∀ A : Prop, A.
Definition Not (A : Prop) := A → False.
Definition True : Prop := Not False.

Definition δ : True := λ z : False, z True z : False.
Definition ω (h : ∀ A B : Prop, seq A B) : False :=
  λ A : Prop, cast True A (h True A) δ : A.
Definition Ω (h : ∀ A B : Prop, seq A B) : False :=
  δ (ω h).

Fail Timeout 1 Eval lazy in Ω.
```

$$\mathsf{cast}\ A\ A\ e\ x \rightsquigarrow_{\delta\beta\iota} x$$

$$\begin{aligned}
\Omega\ h &\rightsquigarrow_{\delta\beta} \delta\ (\omega\ h) \\
&\rightsquigarrow_{\delta\beta} \omega\ h\ \top\ (\omega\ h) \\
&\rightsquigarrow_{\delta\beta} \mathsf{cast}\ \top\ \top\ (h\ \top\ \top)\ \delta\ (\omega\ h) \\
&\rightsquigarrow_{\delta\beta\iota} \delta\ (\omega\ h) \\
&\leftsquigarrow_{\delta\beta} \Omega\ h
\end{aligned}$$

Abel & Coquand nontermination proof: exhibit B

```
Set Definitional UIP.

Inductive seq {A} (a : A) : A → SProp :=
  srefl : seq a a.

Axiom tautext : ∀ (A B : Prop), A → B → seq A B.

Definition True : Prop := ∀ A : Prop, A → A.

Definition cast (A B : Prop) (eq : seq A B) (x : A) : B :=
  match eq with srefl _ ⇒ x end.

Definition id (x : True) : True := x.
Definition δ (z : True) : True := z (True → True) id z.
Definition ω : True := λ (A : Prop) (a : A),
  cast (True → True) A (tautext (True → True) A id a) δ.
Definition Ω : True := δ ω.

Fail Timeout 1 Eval lazy in Ω.
```

$$\text{cast } A\ A\ e\ x \rightsquigarrow_{\delta\beta\iota} x$$

$$\begin{aligned}
\Omega &\rightsquigarrow_{\delta} \delta\ \omega \\
&\rightsquigarrow_{\delta\beta} \omega\ (\top \to \top)\ \text{id}\ \omega \\
&\rightsquigarrow_{\delta\beta} \text{cast}\ (\top \to \top)\ (\top \to \top) \\
&\qquad\qquad (\text{tautext}\ (\top \to \top)\ (\top \to \top)\ \text{id}\ \text{id})\ \delta\ \omega \\
&\rightsquigarrow_{\delta\beta\iota} \delta\ \omega \\
&\leftsquigarrow_{\delta} \Omega
\end{aligned}$$

Honorable mention: positive coinductive types in Coq

Positive coinductive types break subject reduction in Coq:

```
CoInductive Stream : Set := Seq (hd : nat) (tl : Stream).

Definition hd (x : Stream) := let (a, s) := x in a.
Definition tl (x : Stream) := let (a, s) := x in s.

Lemma Stream_eta (s : Stream) : s = Seq (hd s) (tl s).
Proof. Fail reflexivity. destruct s. reflexivity. Qed.

Set Primitive Projections.
CoInductive Stream' : Set := Seq' { hd' : nat; tl' : Stream }.

Lemma Stream'_eta (s : Stream') : s = Seq' (hd' s) (tl' s).
Proof. Fail reflexivity. Fail destruct s. Abort.

(* It is fine to assume the above as an axiom though. *)
```

How can you prove consistency?

Consistency: there is no proof of $\bot$ that the kernel verifies.

How can you prove consistency?

Consistency: there is no proof of $\bot$ that the kernel verifies.

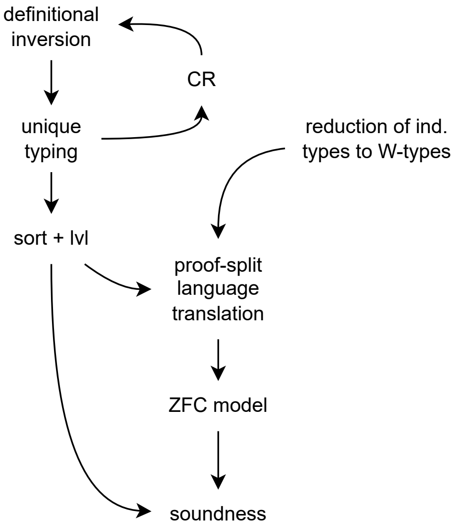
- If you have a known terminating reduction order (so SN/WN) and SR, you can derive consistency if there is no normal form proof of $\bot$.
  - This approach is used by Coquand & Gallier (1990) to prove the consistency of CC.
  - It does not work for Lean since we have seen that some terms have no (weak head) normal form. (Although we have SR for $\vdash$ (so with $\equiv$) as an immediate consequence of the conversion rule.)

How can you prove consistency?

Consistency: there is no proof of $\bot$ that the kernel verifies.

- If you have a known terminating reduction order (so SN/WN) and SR, you can derive consistency if there is no normal form proof of $\bot$.
  - This approach is used by Coquand & Gallier (1990) to prove the consistency of CC.
  - It does not work for Lean since we have seen that some terms have no (weak head) normal form. (Although we have SR for $\vdash$ (so with $\equiv$) as an immediate consequence of the conversion rule.)
- Another option is to construct a model of the theory in a trusted axiomatic framework, *e.g.*, ZFC. Werner (1997) shows equiconsistency of ~CIC with ~ZFC.
  - Carneiro takes this approach for Lean, proving that ZFC + "there are $n+1$ inaccessible cardinals" $\vdash$ Con(Lean with $n+1$ universes).
  - Lean (3) already had a ZFC model adapted from Werner's model, so the reverse direction did not need to be covered. (A model in Lean 4 now also exists.)

Overview


definitional
inversion
CR
unique
typing
reduction of ind.
types to W-types
sort + lvl
proof-split
language
translation
ZFC model
soundness

Lean as sets

- Interpret types ($\Gamma \vdash \alpha$ type) as sets $[\![\Gamma \vdash \alpha]\!]_\gamma$.
  - The γ that you see here is a valuation for the context, *i.e.*, it assigns values to the bindings declared in Γ.
  - Represented as a (dependent) sequence of values.
  - $[\![x_1 : \alpha_1, \ldots, x_n : \alpha_n \vdash x_i]\!]_\gamma = \pi_i(\gamma)$
- Context interpretation: $\gamma \in [\![\Gamma]\!]$:
  - $[\![\cdot]\!] = \{()\}$
  - $[\![\Gamma, x : \alpha]\!] = \Sigma_{\gamma \in [\![\Gamma]\!]} [\![\Gamma \vdash \alpha]\!]_\gamma$
- How do we handle the complexity of inductive types, discern propositions from types and handle universe variables?

Lean as sets

- Interpret types ($\Gamma \vdash \alpha$ type) as sets $[\![\Gamma \vdash \alpha]\!]_\gamma$.
  - The γ that you see here is a valuation for the context, *i.e.*, it assigns values to the bindings declared in Γ.
  - Represented as a (dependent) sequence of values.
  - $[\![x_1 : \alpha_1, \dots, x_n : \alpha_n \vdash x_i]\!]_\gamma = \pi_i(\gamma)$
- Context interpretation: $\gamma \in [\![\Gamma]\!]$:
  - $[\![\cdot]\!] = \{()\}$
  - $[\![\Gamma, x : \alpha]\!] = \Sigma_{\gamma \in [\![\Gamma]\!]} [\![\Gamma \vdash \alpha]\!]_\gamma$
- How do we handle the complexity of inductive types, discern propositions from types and handle universe variables? We don't!

The proof-split language

- Constructs that produce proofs and propositions are separated from those that produce terms and types:

$$\langle e_1\ e_2\rangle = \begin{cases} \langle e_1\rangle_\Gamma\ \langle e_2\rangle_\Gamma & \text{if sort}(\Gamma \vdash e_1) = 0 \\ \langle e_1\rangle_\Gamma \cdot \langle e_2\rangle_\Gamma & \text{if sort}(\Gamma \vdash e_1) \geq 1 \end{cases}$$

$$\langle \lambda x : \alpha.\ e\rangle = \begin{cases} \lambda x : \langle \alpha\rangle_\Gamma.\ \langle e\rangle_{\Gamma, x:a} & \text{if sort}(\Gamma \vdash e) = 0 \\ \Lambda x : \langle \alpha\rangle_\Gamma.\ \langle e\rangle_{\Gamma, x:a} & \text{if sort}(\Gamma \vdash e) \geq 1 \end{cases}$$

(similar for $\forall$: $\mathsf{U}_0 \to \forall$, $\mathsf{U}_1 \to \Pi$)

  This will be very convenient for soundness.
- For simplicity, we also fix a universe level variable valuation here.
- Inductive types have already been translated to $\mathcal{W}$-types + Σ-types (or accessibility-based types for subsingleton types).
- Example: $(\lambda f : \bot.\ \lambda p : \mathbb{P}.\ \downarrow (\mathrm{rec}_{\bot}^{\mathrm{ulift}_0^1\ p} \cdot f)) : \bot \to \forall p : \mathbb{P}.\ p$

Interpretation examples

- All propositions $\Gamma \vdash \alpha : \mathbb{P}$ are truncated to $[\![\Gamma \vdash \alpha]\!]_\gamma \subseteq \{\bullet\}$.
  - This means that all proofs $\Gamma \vdash e : \alpha : \mathbb{P}$ are truncated to $\bullet$.
  - As such, the implications of impredicativity and definitional proof irrelevance (as demonstrated by the Abel & Coquand counterexample to termination) do not bother us.
- To continue the example from last slide:

$$
\begin{aligned}
&[\![\vdash \mathbb{P}]\!]_{()} = [\![\vdash U_0]\!]_{()} = \{\emptyset, \{\bullet\}\} \\
&[\![\vdash \bot]\!]_{()} = \emptyset \\
&[\![\vdash \lambda f.\, \lambda p : \mathbb{P}.\, \downarrow (\mathsf{rec}_{\bot}^{\mathsf{ulift}_0^1\, p} \cdot f)]\!]_{()} = \bullet \\
&[\![\vdash \forall f : \bot.\, \forall p : \mathbb{P}.\, p]\!]_{()} = \{\bullet\} \cap \bigcap_{x \in [\![\vdash \bot]\!]} [\![f : \bot \vdash \forall p : \mathbb{P}.\, p]\!]_{(x)} = \{\bullet\}
\end{aligned}
$$

Soundness

- The general idea of soundness: ensure that everything ends up in the expected set. Four parts to the main theorem (with limit cardinal specifics elided):
  1. If $\Gamma \vdash \alpha : \mathbb{P}$, then $[\![\Gamma \vdash \alpha]\!]_\gamma \subseteq \{\bullet\}$.
  2. If $\Gamma \vdash e : \alpha$ and $\mathsf{lvl}(\Gamma \vdash \alpha) = 0$, then $[\![\Gamma \vdash e]\!]_\gamma = \bullet$.
  3. If $\Gamma \vdash e : \alpha$, then $[\![\Gamma \vdash e]\!]_\gamma \in [\![\Gamma \vdash \alpha]\!]_\gamma$.
  4. If $\Gamma \vdash e \equiv e'$, then for all $\gamma \in [\![\Gamma]\!]$, $[\![\Gamma \vdash e]\!]_\gamma = [\![\Gamma \vdash e']\!]_\gamma$.
- (Note that the example from the last slides satisfies parts 1–3.)
- Simplified final soundness argument: if $\Vdash e : \bot$ then $\vdash e : \bot$, so $\vdash \langle e \rangle_{v,\cdot} : \bot$ (where v sets all universe level variables to zero), but then $[\![\vdash \langle e \rangle]\!]_{()} \in [\![\vdash \bot]\!]_{()} = \emptyset$, a contraction.

Intermezzo: faulty unique typing

Intermezzo: faulty unique typing

- We saw the cycle unique typing $\rightarrow$ CR $\rightarrow$ def. inversion $\rightarrow$ unique typing before.
- It turns out that there is a flaw in how the proof is set up: unique typing is currently merely a conjecture.

Intermezzo: a reasonable proof setup

- Unique typing: if $\Gamma \vdash e : \alpha$ and $\Gamma \vdash e : \beta$, then $\Gamma \vdash \alpha \equiv \beta$.
- CR: if $\Gamma \vdash e : \alpha$ and $e_1 \leftsquigarrow^*_\kappa e \rightsquigarrow^*_\kappa e_2$, then $\exists e_1'\, e_2'.\ e_1 \rightsquigarrow^*_\kappa e_1' \equiv_p e_2' \leftsquigarrow^*_\kappa e_2$.
- Due to the mutual dependency of def. eq. and typing, unique typing and CR depend on each other.
- Idea: we can still set up induction here; we just limit the amount of applications of the conversion rule in the depth of the derivation tree.
  - For a judgement $\Gamma \vdash_n e : \alpha$, every path to a leaf in the derivation tree may see at most n appeals to the conversion rule (roughly).
  - We start with proving unique typing for $\vdash_0$.
  - Then we fix n and prove CR with only $\vdash_n$ typing judgements. (And $\Gamma \vdash_{n+1} e \equiv e'$ judgements, which may only use $\vdash_n$ typing judgements.)
  - Which we then use to prove definitional inversion for $\vdash_{n+1}$, which then again gives us unique typing for $\vdash_{n+1}$.

Intermezzo: a pesky lemma

So far, so good...

- **Lemma 4.6** (Regularity of reductions)**, part** *(4)* (Substitution)**.**
  If $\Gamma, x : \alpha \vdash_{n+1} e_1 \equiv_p e_1'$ *and* $\Gamma \vdash_{n+1} e_2 \equiv_p e_2'$*, then* $\Gamma \vdash_{n+1} e_1[e_2/x] \equiv_p e_1'[e_2'/x]$.

Intermezzo: a pesky lemma

So far, so good... But in requiring $\vdash_n$, a technical lemma breaks:

- **Lemma 4.6** (Regularity of reductions)**, part** *(4)* (Substitution)**.**
  If $\Gamma, x : \alpha \vdash_{n+1} e_1 \equiv_p e_1'$ *and* $\Gamma \vdash_{n+1} e_2 \equiv_p e_2'$, *then* $\Gamma \vdash_{n+1} e_1[e_2/x] \equiv_p e_1'[e_2'/x]$.
- Say we have $e_1 = e_1'$, $e_2 = e_2'$. Then by the reflexivity rule, $\Gamma, x : \alpha \vdash_n e_1 : \beta_1$ and $\Gamma \vdash_n e_2 : \beta_2$ for some β_1, β_2.

Intermezzo: a pesky lemma

So far, so good...

- **Lemma 4.6** (Regularity of reductions)**, part** *(4)* (Substitution)**.**
  If $\Gamma, x : \alpha \vdash_{n+1} e_1 \equiv_p e_1'$ *and* $\Gamma \vdash_{n+1} e_2 \equiv_p e_2'$*, then* $\Gamma \vdash_{n+1} e_1[e_2/x] \equiv_p e_1'[e_2'/x]$*.*
- Say we have $e_1 = e_1'$, $e_2 = e_2'$. Then by the reflexivity rule, $\Gamma, x : \alpha \vdash_n e_1 : \beta_1$ and $\Gamma \vdash_n e_2 : \beta_2$ for some β_1, β_2.
- Now we just need to prove $\Gamma \vdash_n e_1[e_2/x] : \beta_1[\beta_2/x]$.

Intermezzo: a pesky lemma

So far, so good…

- **Lemma 4.6** (Regularity of reductions)**, part** *(4)* (Substitution)**.**
  If $\Gamma, x : \alpha \vdash_{n+1} e_1 \equiv_p e_1'$ *and* $\Gamma \vdash_{n+1} e_2 \equiv_p e_2'$, *then* $\Gamma \vdash_{n+1} e_1[e_2/x] \equiv_p e_1'[e_2'/x]$.
- Say we have $e_1 = e_1'$, $e_2 = e_2'$. Then by the reflexivity rule, $\Gamma, x : \alpha \vdash_n e_1 : \beta_1$ and $\Gamma \vdash_n e_2 : \beta_2$ for some β_1, β_2.
- Now we just need to prove $\Gamma \vdash_n e_1[e_2/x] : \beta_1[\beta_2/x]$. ↯

Intermezzo: a pesky lemma

So far, so good...

- **Lemma 4.6** (Regularity of reductions)**, part** *(4)* (Substitution)**.**
  If $\Gamma, x : \alpha \vdash_{n+1} e_1 \equiv_p e_1'$ *and* $\Gamma \vdash_{n+1} e_2 \equiv_p e_2'$*, then* $\Gamma \vdash_{n+1} e_1[e_2/x] \equiv_p e_1'[e_2'/x]$*.*
- Say we have $e_1 = e_1'$, $e_2 = e_2'$. Then by the reflexivity rule, $\Gamma, x : \alpha \vdash_n e_1 : \beta_1$ and $\Gamma \vdash_n e_2 : \beta_2$ for some β_1, β_2.
- Now we just need to prove $\Gamma \vdash_n e_1[e_2/x] : \beta_1[\beta_2/x]$. $\lightning$
- How bad is this?

Lean4Lean: the state of theory & metatheory

Lean4Lean: the state of theory & metatheory

- The full typing judgement is there and is related with the `Expr` type that Lean uses.
- Since the base theory of Lean has been extended a bit, the soundness proof that we discussed is not applicable without modification.
- Since the proof of the unique typing theorem had an error, it remains to be seen whether it can be salvaged.
  - According to Carneiro, it is also possible to prove soundness without unique typing, but would significantly affect the proof.
- 'The Type Theory of Lean' by Carneiro (2019) left the precise translation of eliminators for inductive types as future work.

The start of formal guarantees: the kernel

The kernel processes all elaborated, ordered definitions, and adds them to the environment if they are well-typed:

$$\text{addDecl} : \text{Environment} \to \text{Declaration} \to \text{Except KernelException Environment}$$

- This interface is very simple; it is not unrealistic to separate the kernel from the rest of the proof assistant or to run a completely independent kernel implementation.
- Lean 3 had such ‘external verifiers’, Lean 4 not.*

Verifying the verifier

- End goal: the type checker is sound, *i.e.*, it does not typecheck any term that is not well-typed according to the theory. (Not considering extra axioms, unsafe definitions etc.)
- Completeness is impossible, certainly w.r.t. the optimal typing judgement $\Gamma \vdash e : \alpha$.
- Before this, all parts of the theory (and some metatheory) need to be implemented.
  - The most important part here is the typing judgement and some additional regularity lemmas, which Carneiro presents in the Lean4Lean paper.
  - So most of the work required seems to already be done, but it remains relatively unclear from the paper what the next required steps for type checker verification are.

Lean4Lean: the state of the verifier

- A reimplementation of the Lean C++ kernel, in Lean.
- Operates on compiled `.olean` files.
- Written so that a proof of soundness should be possible, but not proven correct yet.

	`lean4checker`	`lean4lean`	slowdown
`Lean`	37.01 s	44.61 s	21%
`Batteries`	32.49 s	45.74 s	41%
`Mathlib` (+ `Batteries` + `Lean`)	44.54 min	58.79 min	32%

Table: Comparison of the C++ kernel and Lean4Lean on an i7-1255U, from the latest Lean4Lean preprint

Where does this all leave us?

- For lack of formal proofs, we can not truly trust our proof assistants.
  - For example, CIC itself is well-studied and seems to be consistent. But as recently as 2021, Coq has one 'proof of False' report at least once a year due to implementation bugs. (The kernel is quite complex at ~20kLoC OCaml, ~10kLoC C.)
  - Despite Lean's kernel being much smaller (~6.5kLoC C++) than that of Coq, some soundness bugs still crept in with Lean 4.

Where does this all leave us?

- For lack of formal proofs, we can not truly trust our proof assistants.
  - For example, CIC itself is well-studied and seems to be consistent. But as recently as 2021, Coq has one ‘proof of False’ report at least once a year due to implementation bugs. (The kernel is quite complex at ~20kLoC OCaml, ~10kLoC C.)
  - Despite Lean’s kernel being much smaller (~6.5kLoC C++) than that of Coq, some soundness bugs still crept in with Lean 4.
  - Verification of the kernel is currently the most important task at hand.

Where does this all leave us?

- For lack of formal proofs, we can not truly trust our proof assistants.
  - For example, CIC itself is well-studied and seems to be consistent. But as recently as 2021, Coq has one 'proof of False' report at least once a year due to implementation bugs. (The kernel is quite complex at ~20kLoC OCaml, ~10kLoC C.)
  - Despite Lean's kernel being much smaller (~6.5kLoC C++) than that of Coq, some soundness bugs still crept in with Lean 4.
  - Verification of the kernel is currently the most important task at hand.
  - Lean4Lean is an interesting project, but has very little manpower compared to MetaCoq (especially for such a herculean effort).

Where does this all leave us?

- For lack of formal proofs, we can not truly trust our proof assistants.
  - For example, CIC itself is well-studied and seems to be consistent. But as recently as 2021, Coq has one 'proof of False' report at least once a year due to implementation bugs. (The kernel is quite complex at ~20kLoC OCaml, ~10kLoC C.)
  - Despite Lean's kernel being much smaller (~6.5kLoC C++) than that of Coq, some soundness bugs still crept in with Lean 4.
  - Verification of the kernel is currently the most important task at hand.
  - Lean4Lean is an interesting project, but has very little manpower compared to MetaCoq (especially for such a herculean effort).
- Nevertheless, we have then still not considered most of the rest of the stack, which you must also trust.
- How strong are the guarantees that you require?

Thank you for listening!

Questions?