Proof Assistants for Free*

**Rates may apply*

Pierre-Marie Pédrot

Max Planck Institute for Software Systems

CSEC Kick-off
6th March 2018

CIC: « Constructions dans un monde qui bouge »

CIC, the Calculus of Inductive Constructions.

CIC: « Constructions dans un monde qui bouge »

CIC, the Calculus of Inductive Constructions.

CIC, a very fancy **intuitionistic** **logical system**.

- Not just higher-order logic, not just first-order logic
- First class notion of computation and crazy inductive types

CIC, the Calculus of Inductive Constructions.

CIC, a very fancy **intuitionistic** **logical system**.

- Not just higher-order logic, not just first-order logic
- First class notion of computation and crazy inductive types

CIC, a very powerful **functional** **programming language**.

- Finest types to describe your programs
- No clear phase separation between runtime and compile time

CIC: « Constructions dans un monde qui bouge »

CIC, the Calculus of Inductive Constructions.

CIC, a very fancy **intuitionistic** **logical system**.

- Not just higher-order logic, not just first-order logic
- First class notion of computation and crazy inductive types

CIC, a very powerful **functional** **programming language**.

- Finest types to describe your programs
- No clear phase separation between runtime and compile time

The Pinnacle of the Curry-Howard correspondence

An Effective Object

One implementation to rule them all...

An Effective Object

One implementation to rule them all...

An Effective Object

One implementation to rule them all...

Many big developments using it for computer-checked proofs.

- Mathematics: Four colour theorem, Feit-Thompson, Unimath...
- Computer Science: CompCert, VST, RustBelt...

The CIC Tribe

Actually not quite one single theory.

Several flags tweaking the kernel:

- Impredicative Set
- Type-in-type
- Indices Matter
- Cumulative inductive types
- ...

The CIC Tribe

Actually not quite one single theory.

Several flags tweaking the kernel:

- Impredicative Set
- Type-in-type
- Indices Matter
- Cumulative inductive types
- ...

The Many Calculi of Inductive Constructions.

In the Axiom Jungle

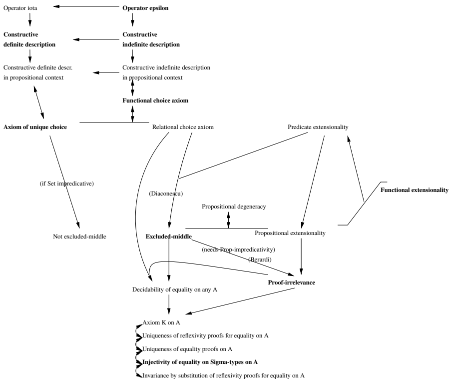
A crazy amount of axioms used in the wild!

In the Axiom Jungle

A crazy amount of axioms used in the wild!

The *claſſical ſet-theory* pole:

- Excluded middle, UIP, choice



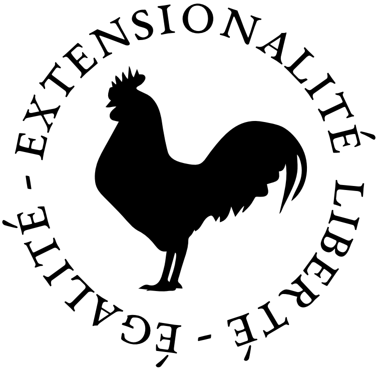
A crazy amount of axioms used in the wild!

The classical set-theory pole:

- Excluded middle, UIP, choice

The Extensional pole:

- Funext, Propext, Bisim-is-eq



A crazy amount of axioms used in the wild!

The classical set-theory pole:

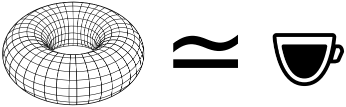
- Excluded middle, UIP, choice

The EXTENSIONAL pole:

- Funext, Propext, Bisim-is-eq

The univalent pole:

- Univalence, what else?



« A mathematician is a device for turning toruses into equalities (up to homotopy). »

In the Axiom Jungle

A crazy amount of axioms used in the wild!

The classical set-theory pole:

- Excluded middle, UIP, choice

The EXTENSIONAL pole:

- Funext, Propext, Bisim-is-eq

The univalent pole:

- Univalence, what else?

The $\varepsilon\chi o\tau\iota c$ pole:

- Anti-classical axioms (???)

In the Axiom Jungle

A crazy amount of axioms used in the wild!

The classical set-theory pole:

- Excluded middle, UIP, choice

The EXTENSIONAL pole:

- Funext, Propext, Bisim-is-eq

The univalent pole:

- Univalence, what else?

The $\varepsilon\chi o\tau\iota c$ pole:

- Anti-classical axioms (???)

Varying degrees of compatibility.

Reality Check

Theorem 0

Axioms Suck.

Reality Check

Theorem 0

Axioms Suck.

Proof.

- They break computation (and thus canonicity).
- They are hard to justify.
- They might be incompatible with one another.

□

Look ma, no Axioms

Alternative route to axioms: **implement** a new type theory.

Examples: Cubical, F*...

Look ma, no Axioms

Alternative route to axioms: **implement** a new type theory.

Examples: Cubical, F*...

Pro

- Computational by construction (hopefully)
- Tailored for a specific theory

Look ma, no Axioms

Alternative route to axioms: **implement** a new type theory.

Examples: Cubical, F*...

Pro

- Computational by construction (hopefully)
- Tailored for a specific theory

Con

- Requires a new proof of soundness (... *cough*... right, F*? *cough*...)
- Implementation task may be daunting (including bugs)
- Yet-another-language: say farewell to libraries, tools, community...

Summary of the Problem

Different users have different needs.

« From each according to his ability, to each according to his needs. »

(Excessive) Fragmentation of proof assistants is harmful.

« Divide et impera. »

Summary of the Problem

Different users have different needs.

« From each according to his ability, to each according to his needs. »

(Excessive) Fragmentation of proof assistants is harmful.

« Divide et impera. »

Are we thus doomed?

Teasing

In this talk, I'd like to advocate for a third way.

One implementation to rule them all...

Teasing

In this talk, I'd like to advocate for a third way.

~~One imple[illegible] them all...~~

Teasing

In this talk, I'd like to advocate for a third way.

~~One implementation to rule them all...~~

One **backend** implementation to rule them all!

In this talk, I'd like to advocate for a third way.

~~One implementation to rule them all...~~

One **backend** implementation to rule them all!

via

Syntactic Models

From Hell

Semantics of type theory have a fame of being horribly complex.

From Hell

Semantics of type theory have a fame of being horribly complex.

I won't lie: **they are**. But part of this fame is due to its usual models.

From Hell

Semantics of type theory have a fame of being horribly complex.

I won't lie: **they are**. But part of this fame is due to its usual models.

Roughly three families of models:

- The **set-theoretical** model and its variants
- Several **realizability** models
- A gazillion of **categorical** models

Let's review them quickly!

The Set-Theoretical Model

Because Sets are a (crappy) type theory.

The Set-Theoretical Model

Because Sets are a (crappy) type theory.

Interpret everything as sets and expect $\vdash_{\mathrm{CIC}} M : A \Rightarrow \vdash_{\mathrm{ZFC}} [M] \in [A]$.

$$[\Pi x : A.\, B] \equiv \left\{ f \in [A] \to_{\mathrm{ZFC}} \bigcup_{x \in [A]} [B](x) \;\middle|\; \forall x \in [A].\, f(x) \in [B](x) \right\}$$

The Set-Theoretical Model

Because Sets are a (crappy) type theory.

Interpret everything as sets and expect $\vdash_{\mathrm{CIC}} M : A \Rightarrow \vdash_{\mathrm{ZFC}} [M] \in [A]$.

$$[\Pi x : A.\, B] \equiv \left\{ f \in [A] \to_{\mathrm{ZFC}} \bigcup_{x \in [A]} [B](x) \;\middle|\; \forall x \in [A].\, f(x) \in [B](x) \right\}$$

Pro

- Well-known and trusted target
- Imports ZFC properties.

The Set-Theoretical Model

Because Sets are a (crappy) type theory.

Interpret everything as sets and expect $\vdash_{\mathrm{CIC}} M : A \Rightarrow \vdash_{\mathrm{ZFC}} [M] \in [A]$.

$$[\Pi x : A.\, B] \equiv \left\{ f \in [A] \to_{\mathrm{ZFC}} \bigcup_{x \in [A]} [B](x) \;\middle|\; \forall x \in [A].\, f(x) \in [B](x) \right\}$$

Pro

- Well-known and trusted target
- Imports ZFC properties.

Con

- Forego syntax, computation and decidability
- Imports ZFC properties.

The Realizability Models

Construct programs that respect properties.

The Realizability Models

Construct programs that respect properties.

- Terms $M \rightsquigarrow$ programs $[M]$ (variable languages as a target)
- Types $A \rightsquigarrow$ meta-theoretical predicates $[\![A]\!]$
- $\vdash_{\mathrm{CIC}} M : A \quad \Rightarrow \quad [M] \in [\![A]\!]$

$$[\![\Pi x : A.\, B]\!] \equiv \{f \in \Lambda \mid \forall x \in [\![A]\!].\, \mathtt{eval}(f, x) \in [\![B]\!](x)\}$$

The Realizability Models

Construct programs that respect properties.

- Terms $M \rightsquigarrow$ programs $[M]$ (variable languages as a target)
- Types $A \rightsquigarrow$ meta-theoretical predicates $[\![A]\!]$
- $\vdash_{\mathrm{CIC}} M : A \quad \Rightarrow \quad [M] \in [\![A]\!]$

$$[\![\Pi x : A.\, B]\!] \equiv \{f \in \Lambda \mid \forall x \in [\![A]\!].\, \texttt{eval}(f, x) \in [\![B]\!](x)\}$$

Pro

- Some preservation of syntax and computability

The Realizability Models

Construct programs that respect properties.

- Terms $M \rightsquigarrow$ programs $[M]$ (variable languages as a target)
- Types $A \rightsquigarrow$ meta-theoretical predicates $[\![A]\!]$
- $\vdash_{\mathrm{CIC}} M : A \quad \Rightarrow \quad [M] \in [\![A]\!]$

$$[\![\Pi x : A.\, B]\!] \equiv \{f \in \Lambda \mid \forall x \in [\![A]\!].\, \mathtt{eval}(f, x) \in [\![B]\!](x)\}$$

Pro

- Some preservation of syntax and computability

Con

- Usually crazily undecidable
- Meta-theory can be arbitrary crap, including ZFC

Abstract (nonsense) description of type theory.

The Categorical Models

Abstract (nonsense) description of type theory.

Rephrase the rules of CIC in a categorical way.

The Categorical Models

Abstract (nonsense) description of type theory.

Rephrase the rules of CIC in a categorical way.

Pro

- Very abstract and subsumes both previous examples
- Somewhat "easier" to show some structure is a model of TT

The Categorical Models

Abstract (nonsense) description of type theory.

Rephrase the rules of CIC in a categorical way.

Pro

- Very abstract and subsumes both previous examples
- Somewhat "easier" to show some structure is a model of TT

Con

- Same limitations as the previous examples
- Mostly useless to actually construct a model
- Yet another syntax, usually arcane and ill-fitted

What's The Matter

Assuming we pick a specific model, what do we do with it?

What's The Matter

Assuming we pick a specific model, what do we do with it?

Hopefully it has a more refined content!

In particular, you can show that an axiom hold in this model.

What's The Matter

Assuming we pick a specific model, what do we do with it?

Hopefully it has a more refined content!

In particular, you can show that an axiom hold in this model.

For instance, in **Set**:

$$[\mathrm{Prop}] \equiv \{\varnothing, \{\varnothing\}\}$$

so in there you can inhabit e.g.

$$\texttt{prop\_ext} : \Pi(A\,B : \mathrm{Prop}).\,(A \leftrightarrow B) \rightarrow A = B$$

$$\texttt{em} : \Pi(A : \mathrm{Prop}).\,A + \neg A$$

What is a model?

- Takes syntax as input.
- Interprets it into some low-level language.
- Must preserve the meaning of the source.
- Refines the behaviour of under-specified structures.

What is a model?

- Takes syntax as input.
- Interprets it into some low-level language.
- Must preserve the meaning of the source.
- Refines the behaviour of under-specified structures.

Luckily we're computer scientists in here.

What is a model?

- Takes syntax as input.
- Interprets it into some low-level language.
- Must preserve the meaning of the source.
- Refines the behaviour of under-specified structures.

Luckily we're computer scientists in here.

« Oh yes, we call that a *compiler*... »

(Thanks, Curry-Howard!)

Curry-Howard Orthodoxy

Let's look at what Curry-Howard provides in simpler settings.

Program Translations $\Leftrightarrow$ Logical Interpretations

Curry-Howard Orthodoxy

Let's look at what Curry-Howard provides in simpler settings.

Program Translations $\Leftrightarrow$ Logical Interpretations

On the **programming** side, enrich the language by program translation.

- Monadic style à la Haskell
- Compilation of higher-level constructs down to assembly

Curry-Howard Orthodoxy

Let's look at what Curry-Howard provides in simpler settings.

Program Translations $\Leftrightarrow$ Logical Interpretations

On the **programming** side, enrich the language by program translation.

- Monadic style à la Haskell
- Compilation of higher-level constructs down to assembly

On the **logic** side, extend expressivity through proof interpretation.

- Double-negation $\Rightarrow$ classical logic (`callcc`)
- Friedman's trick $\Rightarrow$ Markov's rule (exceptions)
- Forcing $\Rightarrow$ $\neg\mathrm{CH}$ (global monotonous cell)

Syntactic Models

Let us do the same thing with CIC: build **syntactic models**.

Let us do the same thing with CIC: build **syntactic models**.

We take the following act of faith for granted.

CIC is.

Let us do the same thing with CIC: build **syntactic models**.

We take the following act of faith for granted.

CIC is.

Not caring for its soundness, implementation, whatever. It just is.

Do everything by interpreting the new theories relatively to this foundation!

Suppress technical and cognitive burden by lowering impedance mismatch.

Syntactic Models II

Step 0: Fix a theory $\mathcal{T}$ as close as possible* to CIC, ideally $\mathrm{CIC} \subseteq \mathcal{T}$.

Syntactic Models II

Step 0: Fix a theory $\mathcal{T}$ as close as possible* to CIC, ideally $\mathrm{CIC} \subseteq \mathcal{T}$.

Step 1: Define $[\cdot]$ on the syntax of $\mathcal{T}$ and derive $[\![\cdot]\!]$ from it s.t.

$$\vdash_{\mathcal{T}} M : A \qquad \text{implies} \qquad \vdash_{\mathrm{CIC}} [M] : [\![A]\!]$$

Syntactic Models II

Step 0: Fix a theory $\mathcal{T}$ as close as possible* to CIC, ideally $\mathrm{CIC} \subseteq \mathcal{T}$.

Step 1: Define $[\cdot]$ on the syntax of $\mathcal{T}$ and derive $[\![\cdot]\!]$ from it s.t.

$$\vdash_{\mathcal{T}} M : A \qquad \text{implies} \qquad \vdash_{\mathrm{CIC}} [M] : [\![A]\!]$$

Step 2: Flip views and actually pose

$$\vdash_{\mathcal{T}} M : A \qquad \triangleq \qquad \vdash_{\mathrm{CIC}} [M] : [\![A]\!]$$

Syntactic Models II

Step 0: Fix a theory $\mathcal{T}$ as close as possible* to CIC, ideally $\mathrm{CIC} \subseteq \mathcal{T}$.

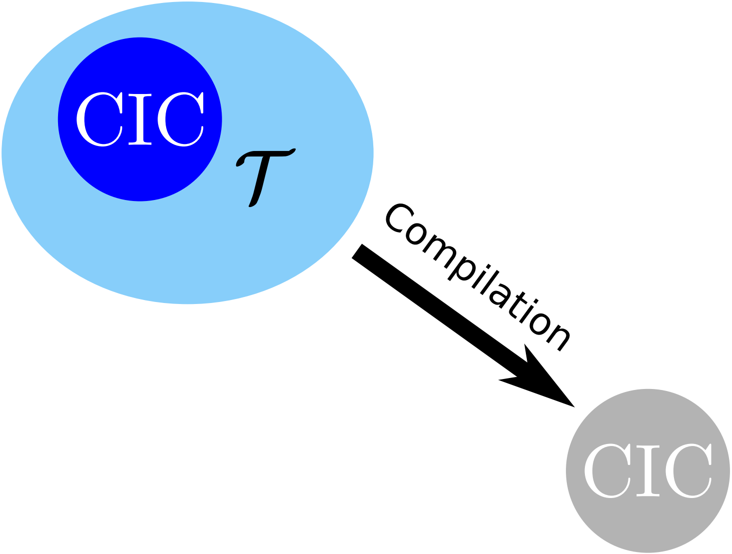
Step 1: Define $[\cdot]$ on the syntax of $\mathcal{T}$ and derive $[\![\cdot]\!]$ from it s.t.

$$\vdash_{\mathcal{T}} M : A \qquad \text{implies} \qquad \vdash_{\mathrm{CIC}} [M] : [\![A]\!]$$

Step 2: Flip views and actually pose

$$\vdash_{\mathcal{T}} M : A \qquad \triangleq \qquad \vdash_{\mathrm{CIC}} [M] : [\![A]\!]$$

Step 3: Expand $\mathcal{T}$ by going down to the *CIC assembly language*, implementing new terms given by the $[\cdot]$ translation.



« CIC, the LLVM of Type Theory »

Syntactic Models III

Obviously, that's subtle.

- The translation $[\cdot]$ must preserve typing (not easy)
- In particular, it must preserve conversion (even worse)

Syntactic Models III

Obviously, that's subtle.

- The translation $[\cdot]$ must preserve typing (not easy)
- In particular, it must preserve conversion (even worse)

Yet, a lot of nice consequences.

- Does not require non-type-theoretical foundations (*monism*)
- **Can be implemented in Coq** (*software monism*)
- Easy to show (relative) consistency, look at $[\![\texttt{False}]\!]$
- Inherit properties from CIC: computationality, decidability...

In Practice: Aknowledge the Existing

In Coq, first require the plugin implementing the desired model.

```
Require Import ExtendCoq.
```

In Practice: Aknowledge the Existing

In Coq, first require the plugin implementing the desired model.

```
Require Import ExtendCoq.
```

Soundness means that any Coq proof can be translated automatically.

```
ExtendCoq Translate cool_theorem.
```

In Practice: Aknowledge the Existing

In Coq, first require the plugin implementing the desired model.

```
Require Import ExtendCoq.
```

Soundness means that any Coq proof can be translated automatically.

```
ExtendCoq Translate cool_theorem.
```

Assuming `cool_theorem` : T, this command:

- defines `cool_theorem`$^{\bullet}$: $[\![T]\!]$
- register the fact that $[$`cool_theorem`$] :=$ `cool_theorem`$^{\bullet}$

Thus any later use of `cool_theorem` in a translated term will be automatically turned into `cool_theorem`$^{\bullet}$.

In Practice: Enlarge Your Theory

The interest of this approach lies in the following command.

```
ExtendCoq Definition new : N.
```

In Practice: Enlarge Your Theory

The interest of this approach lies in the following command.

`ExtendCoq Definition new : N.`

This opens a goal $[\![\mathtt{N}]\!]$ you have to prove.

When the proof is finished:

1. an axiom $\mathtt{new} : \mathtt{N}$ is added;
2. a term $\mathtt{new}^{\bullet} : [\![\mathtt{N}]\!]$ is defined with the proof;
3. the translation $[\![\mathtt{new}]\!] := \mathtt{new}^{\bullet}$ is registered.

In Practice: Dirty Tricks

In general, $[\![\mathtt{N}]\!]$ is some kind of mildly unreadable type that is crazy enough so that it has more inhabitants than N.

```
forall
  (A : Type)
  (B : nat → Type),
  (A →
    { n : nat & B n }) →
  { n : nat &
    A → B n }
```

```
forall
  (A : El Type°)
  (B : nat° → El Type°),
  (El A →
    sigT° (TypeVal nat° nat#) (fun n : nat° => B n)) →
  sigT° (TypeVal nat° nat#)
    (fun n : nat° => Prod° (El A) (fun _ : El A => B n))
```

With a bit of practice, you can usually make sense of it though.

Back to Marketing

On-the-fly compilation of the extended theory to Coq!

No more axioms!

Your type-theoretic desires made true!

BEFORE

« Holy Celestial Teapot! »

AFTER

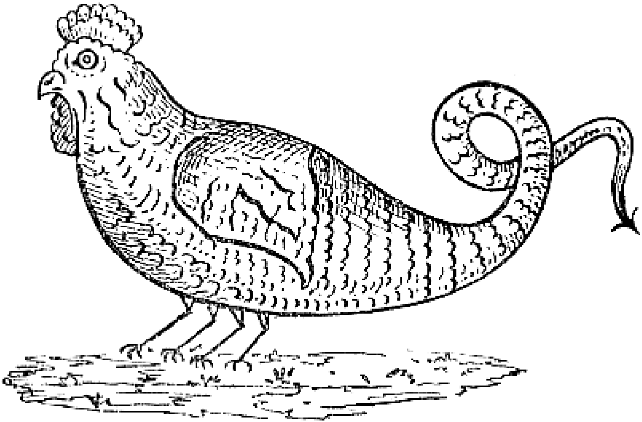
« Stock photos do not experience existential dread. »

**Text and pictures not contractually binding.*

A Few Examples

In the remainder of the talk, I'll describe two simple examples.

- Mostly pedagogical
- Not really interesting in practice
- Still funny to mess with CIC



Ex 1. Intensional Types, a.k.a. **Dynamically Typed CIC**

Intensional Types

The intensional types translation extends type theory with

$$
\begin{array}{lcl}
\texttt{flip} & : & \square \to \square \\
\texttt{flip\_equiv} & : & \Pi(A : \square).\, \texttt{flip}\ A \cong A \\
\texttt{flip\_neq} & : & \Pi(A : \square).\, \texttt{flip}\ A \neq A
\end{array}
$$

Intensional Types

The intensional types translation extends type theory with

$$
\begin{array}{lcl}
\texttt{flip} & : & \square \to \square \\
\texttt{flip\_equiv} & : & \Pi(A : \square).\, \texttt{flip}\ A \cong A \\
\texttt{flip\_neq} & : & \Pi(A : \square).\, \texttt{flip}\ A \neq A
\end{array}
$$

This breaks amongst other things univalence...

The Intensional Types Implementation

Intuitively:

- Translate $A : \square$ into $[A] : \square \times \mathbb{B}$
- Translate $M : A$ into $[M] : [A].\pi_1$

The Intensional Types Implementation

Intuitively:

- Translate $A : \square$ into $[A] : \square \times \mathbb{B}$
- Translate $M : A$ into $[M] : [A].\pi_1$

$$
\begin{array}{lcl}
[\![A]\!] & \equiv & [A].\pi_1 \\
[\square] & \equiv & (\square \times \mathbb{B}, \texttt{true}) \\
[\Pi x : A.\, B] & \equiv & (\Pi x : [\![A]\!].\, [\![B]\!], \texttt{true}) \\
[x] & \equiv & x \\
[M\ N] & \equiv & [M]\ [N] \\
[\lambda x : A.\, M] & \equiv & \lambda x : [\![A]\!].\, [M]
\end{array}
$$

Types contain a boolean not used for their inhabitants!

The Intensional Types Implementation

Intuitively:

- Translate $A : \square$ into $[A] : \square \times \mathbb{B}$
- Translate $M : A$ into $[M] : [A].\pi_1$

$$
\begin{array}{lcl}
[\![A]\!] & \equiv & [A].\pi_1 \\
[\square] & \equiv & (\square \times \mathbb{B}, \mathtt{true}) \\
[\Pi x : A.\, B] & \equiv & (\Pi x : [\![A]\!].\, [\![B]\!], \mathtt{true}) \\
[x] & \equiv & x \\
[M\ N] & \equiv & [M]\ [N] \\
[\lambda x : A.\, M] & \equiv & \lambda x : [\![A]\!].\, [M]
\end{array}
$$

Types contain a boolean not used for their inhabitants!

Soundness

If $\vec{x} : \Gamma \vdash M : A$ then $\vec{x} : [\![\Gamma]\!] \vdash [M] : [\![A]\!]$.

Extending the Intensional Types

Let's define the new operations obtained through the translation.

$$
\begin{array}{lcl}
[\texttt{flip}] & : & [\![\square \to \square]\!] \\
[\texttt{flip}] & : & \square \times \mathbb{B} \to \square \times \mathbb{B} \\
[\texttt{flip}] & \equiv & \lambda(A, b).\,(A, \texttt{negb}\ b) \\
\\
[\texttt{flip\_equiv}] & : & [\![\Pi A : \square.\, \texttt{flip}\ A \cong A]\!] \\
[\texttt{flip\_equiv}] & \equiv & \ldots \\
\\
[\texttt{flip\_neq}] & : & [\![\Pi A : \square.\, \texttt{flip}\ A \neq A]\!] \\
[\texttt{flip\_neq}] & : & \Pi A : \square \times \mathbb{B}.\, [\texttt{flip}]\ A \neq A \\
[\texttt{flip\_equiv}] & \equiv & \ldots
\end{array}
$$

- $[\![\texttt{flip}\ A]\!] \equiv [\![A]\!]$
- And isomorphism only depends on $[\![A]\!]$
- But (intensional) equality observes the boolean...

Basilisk for Realz

This one example is not very interesting.

Basilisk for Realz

This one example is not very interesting.

You can do much better: a real mix of Python and Coq!

Basilisk for Realz

This one example is not very interesting.

You can do much better: a real mix of Python and Coq!

- Assuming the target theory features induction-recursion
- Represent (source) types by their code
- This gives a real type-quote function in the source theory

$$
\begin{array}{ll}
\texttt{type\_rect}: & \Pi(P : \square \to \square). \\
& \quad P\ \square \to \\
& \quad (\Pi(A : \square)\,(B : A \to \square).\ P\ A \to (\Pi x : A.\ P\ (B\ x)) \to \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad P\ (\Pi x : A.\ B)) \to \\
& \quad P\ \mathbb{N} \to \\
& \quad \ldots \to \\
& \Pi(A : \square).\ P\ A
\end{array}
$$

Basilisk for Realz

This one example is not very interesting.

You can do much better: a real mix of Python and Coq!

- Assuming the target theory features induction-recursion
- Represent (source) types by their code
- This gives a real type-quote function in the source theory

$$
\begin{array}{ll}
\texttt{type\_rect}: & \Pi(P : \square \to \square). \\
& \quad P\ \square \to \\
& \quad (\Pi(A : \square)\,(B : A \to \square).\ P\ A \to (\Pi x : A.\ P\ (B\ x)) \to \\
& \qquad\qquad\qquad\qquad\qquad\qquad P\ (\Pi x : A.\ B)) \to \\
& \quad P\ \mathbb{N} \to \\
& \quad \ldots \to \\
& \Pi(A : \square).\ P\ A
\end{array}
$$

Coq is compatible with dynamic types!!!

Ex 2. The reader translation, a.k.a. **Baby Forcing**

The Reader Translation

The reader translation extends type theory with

$$\begin{array}{ccl} \mathbb{R} & : & \square \\ \mathtt{read} & : & \mathbb{R} \\ \mathtt{into} & : & \square \to \mathbb{R} \to \square \\ \mathtt{enter}_A & : & A \to \Pi r : \mathbb{R}.\, \mathtt{into}\ A\ r \end{array}$$

satisfying a few expected definitional equations.

The Reader Translation

The reader translation extends type theory with

$$\begin{array}{rcl} \mathbb{R} & : & \square \\ \texttt{read} & : & \mathbb{R} \\ \texttt{into} & : & \square \to \mathbb{R} \to \square \\ \texttt{enter}_A & : & A \to \Pi r : \mathbb{R}.\, \texttt{into}\ A\ r \end{array}$$

satisfying a few expected definitional equations.

The `into` function has unfoldings on type formers:

$$\begin{array}{lcl} \texttt{into}\ (\Pi x : A.\, B)\ r & \equiv & \Pi x : A.\, \texttt{into}\ B\ r \\ \texttt{into}\ \square\ r & \equiv & \square \\ \ldots & & \end{array}$$

and it is somewhat redundant:

$$\texttt{enter}_{\square}\ A\ r \quad \equiv \quad \texttt{into}\ A\ r$$

The Reader Implementation

Assuming $r : \mathbb{R}$, intuitively:

- Translate $A : \square$ into $[A]_r : \square$
- Translate $M : A$ into $[M]_r : [A]_r$

The Reader Implementation

Assuming $r : \mathbb{R}$, intuitively:

- Translate $A : \square$ into $[A]_r : \square$
- Translate $M : A$ into $[M]_r : [A]_r$

$$\begin{array}{lcl}
[\square]_r & \equiv & \square \\
[\Pi x : A.\, B]_r & \equiv & \Pi x : (\Pi s : \mathbb{R}.\, [A]_s).\, [B]_r \\
[x]_r & \equiv & x\ r \\
[M\ N]_r & \equiv & [M]_r\ (\lambda s : \mathbb{R}.\, [N]_s) \\
[\lambda x : A.\, M]_r & \equiv & \lambda x : (\Pi s : \mathbb{R}.\, [A]_s).\, [M]_r
\end{array}$$

All variables are thunked w.r.t. $\mathbb{R}$!

The Reader Implementation

Assuming $r : \mathbb{R}$, intuitively:

- Translate $A : \Box$ into $[A]_r : \Box$
- Translate $M : A$ into $[M]_r : [A]_r$

$$
\begin{array}{lcl}
[\Box]_r & \equiv & \Box \\
[\Pi x : A.\, B]_r & \equiv & \Pi x : (\Pi s : \mathbb{R}.\, [A]_s).\, [B]_r \\
[x]_r & \equiv & x\ r \\
[M\ N]_r & \equiv & [M]_r\ (\lambda s : \mathbb{R}.\, [N]_s) \\
[\lambda x : A.\, M]_r & \equiv & \lambda x : (\Pi s : \mathbb{R}.\, [A]_s).\, [M]_r
\end{array}
$$

All variables are thunked w.r.t. $\mathbb{R}$!

Soundness

If $\vec{x} : \Gamma \vdash M : A$ then $r : \mathbb{R}, \vec{x} : (\Pi s : \mathbb{R}.\, [\Gamma]_s) \vdash [M]_r : [A]_r$.

Extending the Reader

One can easily define the new operations through the translation.

$$
\begin{array}{lcl}
[\mathbb{R}]_r & : & [\square]_r \\
[\mathbb{R}]_r & : & \square \\
[\mathbb{R}]_r & \equiv & \mathbb{R} \\
\\
[\texttt{read}]_r & : & [\mathbb{R}]_r \\
[\texttt{read}]_r & : & \mathbb{R} \\
[\texttt{read}]_r & \equiv & r \\
\\
[\texttt{into}]_r & : & [\square \to \mathbb{R} \to \square]_r \\
[\texttt{into}]_r & : & (\mathbb{R} \to \square) \to (\mathbb{R} \to \mathbb{R}) \to \square \\
[\texttt{into}]_r & \equiv & \lambda(A : \mathbb{R} \to \square)(\varphi : \mathbb{R} \to \mathbb{R}).\, A\ (\varphi\ r) \\
\\
[\texttt{enter}_A]_r & : & [A \to \Pi s : \mathbb{R}.\, \texttt{into}\ A\ s]_r \\
[\texttt{enter}_A]_r & : & (\Pi s : \mathbb{R}.\, A\ s) \to \Pi(\varphi : \mathbb{R} \to \mathbb{R}).\, A\ (\varphi\ r) \\
[\texttt{enter}_A]_r & \equiv & \lambda(x : \Pi s : \mathbb{R}.\, A\ s)(\varphi : \mathbb{R} \to \mathbb{R}).\, x\ (\varphi\ r)
\end{array}
$$

Utter Lies

The reader translation suffers from one serious limitation!

Utter Lies

The reader translation suffers from one serious limitation!

I won't describe it here. Come back on Thursday!

Spoiler: this is an effect, and that plays bad with dependent elimination.

I cleverly did not describe the translation on the inductive fragment.

More generally

Syntactic models were introduced by M. Hoffmann...

There have been quite a few around since.

Model	Source*	Implements
Parametricity	no `Prop`	Parametricity
Type-intensionality	no `Prop`	Dynamic typing
Reader	BTT	Proof-relevant Axiom
Forcing	BTT	step indexing, nominal reasoning, ...
Weaning	BTT	many effects
Exceptional	no sing. elim.	exceptions (inconsistent)
Exceptional (interm.)	no sing. elim.	Markov's rule
Param. Exceptional	no `Prop`	IP, ...
Extraction	CIC	???
Iso-Parametricity	???	Automatic transfer of properties
Intuitionistic CPS	only `Prop`	???
Dialectica	no `Prop`	Weak MP, ...

The Ugly

To be fair, syntactic models have a few limitations.

- Pretty hard to come up with such models
- Vanilla CIC doesn't seem ideal as a target
- Implementation issues
- For now still rather simple extensions
- Certain complex models seem out of reach (notably univalence)

Still, I argue that they are damn cool.

Thanks for your attention.