Une Théorie des Types qui fait de l'effet

Pierre-Marie Pédrot

Gallinette (Inria Rennes-à-Nantes)

JFLA 2019

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: « 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

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**.

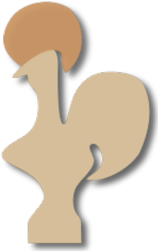
- Finest types to describe your programs
- No clear phase separation between runtime and compile time

The Pinnacle of the Curry-Howard correspondence

An Effective Object

One implementation to rule them all...

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...

What are dependent types?

What are dependent types?

Trivial : *types depend on programs*

$$\texttt{hd} : \Pi(A : \texttt{Type})\,(n : \mathbb{N}).\,\texttt{vect}\ A\ (n + 1) \rightarrow A$$

What are dependent types?

Trivial : *types depend on programs*

$$\mathtt{hd} : \Pi(A : \mathtt{Type})\,(n : \mathbb{N}).\,\mathtt{vect}\ A\ (n+1) \to A$$

Important : n is quantified over **terms** of the language.

(*Chassez cette méta-théorie que je ne saurais voir !*)

Less trivial: *types depend on computation*

let

$$\begin{array}{lcl} \texttt{init} & : & \Pi n : \mathbb{N}.\, \texttt{vect}\ \mathbb{N}\ n \\ \texttt{init}\ n & := & [0; \ldots; n-1] \end{array}$$

and

$$\begin{array}{lcl} g & : & \texttt{option}\ \mathbb{N} \to \mathbb{N} \\ g\ \texttt{None} & := & 0 \\ g\ (\texttt{Some}\ n) & := & n+1 \end{array}$$

Slightly non-trivial

Less trivial: *types depend on computation*

let

$$\begin{array}{lcl} \mathtt{init} & : & \Pi n : \mathbb{N}.\, \mathtt{vect}\ \mathbb{N}\ n \\ \mathtt{init}\ n & := & [0; \ldots; n-1] \end{array}$$

and

$$\begin{array}{lcl} g & : & \mathtt{option}\ \mathbb{N} \to \mathbb{N} \\ g\ \mathtt{None} & := & 0 \\ g\ (\mathtt{Some}\ n) & := & n+1 \end{array}$$

then

$$\mathtt{init}\ (g\ (\mathtt{Some}\ 42)) : \mathtt{vect}\ \mathbb{N}\ 43$$

because

$$g\ (\mathtt{Some}\ 42) \quad \equiv \quad 43$$

Slightly non-trivial

Less trivial: *types depend on computation*

let

$$\begin{array}{lcl} \texttt{init} & : & \Pi n : \mathbb{N}.\, \texttt{vect}\ \mathbb{N}\ n \\ \texttt{init}\ n & := & [0; \ldots; n-1] \end{array}$$

and

$$\begin{array}{lcl} g & : & \texttt{option}\ \mathbb{N} \to \mathbb{N} \\ g\ \texttt{None} & := & 0 \\ g\ (\texttt{Some}\ n) & := & n+1 \end{array}$$

then

$$\texttt{init}\ (g\ (\texttt{Some}\ 42)) : \texttt{vect}\ \mathbb{N}\ 43$$

because

$$g\ (\texttt{Some}\ 42) \quad \equiv \quad 43$$

It computes everywhere!

WTF CIC?

What is going on: *Ça dépend, ça dépasse.*

$\lambda b : \mathbb{B}.\, \texttt{if}\ b\ \texttt{then}\ 0\ \texttt{else}\ \texttt{true}$

WTF CIC?

What is going on: *Ça dépend, ça dépasse.*

$$\lambda b : \mathbb{B}.\,\texttt{if}\ b\ \texttt{then}\ 0\ \texttt{else}\ \texttt{true}\ \ :\ \ \Pi b : \mathbb{B}.\,\texttt{if}\ b\ \texttt{then}\ \mathbb{N}\ \texttt{else}\ \mathbb{B}$$

WTF CIC?

What is going on: *Ça dépend, ça dépasse.*

$$\lambda b : \mathbb{B}.\, \texttt{if}\ b\ \texttt{then}\ 0\ \texttt{else}\ \texttt{true} \quad : \quad \Pi b : \mathbb{B}.\, \texttt{if}\ b\ \texttt{then}\ \mathbb{N}\ \texttt{else}\ \mathbb{B}$$

MER IL ET FOU

You will have a little more *mind-blowing*?

Proofs are *relevant*.

"Any list can be quotiented by permutations."

- Quicksort
- Mergesort
- Bogosort

You will have a little more *mind-blowing*?

Proofs are *relevant*.

"Any list can be quotiented by permutations."

- Quicksort
- Mergesort
- Bogosort

Proofs are first-class objects.

What about $\Pi(A : \mathtt{Type}).\, \Pi(x : A).\, \Pi(p\, q : x = x).\, p = q$?

You will have a little more *mind-blowing*?

Proofs are *relevant*.

"Any list can be quotiented by permutations."

- Quicksort
- Mergesort
- Bogosort

Proofs are first-class objects.

What about $\Pi(A : \mathtt{Type}).\, \Pi(x : A).\, \Pi(p\, q : x = x).\, p = q$?

Un indice, chez vous

This innocent-looking question dragged a Fields medalist into type theory.

The dependent product : the heroin of type theory

The dependent product, a generalization of arrow types?

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma, x : A \vdash B : \texttt{Type}}{\Gamma \vdash \Pi x : A.\, B : \texttt{Type}}$$

The dependent product : the heroin of type theory

The dependent product, a generalization of arrow types?

$$\frac{\Gamma \vdash A : \mathtt{Type} \qquad \Gamma, x : A \vdash B : \mathtt{Type}}{\Gamma \vdash \Pi x : A.\, B : \mathtt{Type}}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.\, t : \Pi x : A.\, B}$$

$$\frac{\Gamma \vdash t : \Pi x : A.\, B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B\{x := u\}}$$

$$(\lambda x : A.\, t)\, u \quad \equiv \quad t\{x := u\}$$

The dependent product, a generalization of arrow types?

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma, x : A \vdash B : \texttt{Type}}{\Gamma \vdash \Pi x : A.\, B : \texttt{Type}}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.\, t : \Pi x : A.\, B}$$

$$\frac{\Gamma \vdash t : \Pi x : A.\, B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B\{x := u\}}$$

$$(\lambda x : A.\, t)\, u \quad \equiv \quad t\{x := u\}$$

Answer: Yes.

$$A \to B \quad \equiv \quad \Pi(\_ : A).\, B$$

Dependent elimination: mind the withdrawal

The other *killer feature* of dependent types.

$$\frac{\Gamma \vdash t_1 : A}{\Gamma \vdash \texttt{inl}\ t_1 : A + B} \qquad \frac{\Gamma \vdash t_2 : B}{\Gamma \vdash \texttt{inr}\ t_2 : A + B}$$

$$\frac{\Gamma, s : A + B \vdash P : \texttt{Type} \quad \Gamma \vdash p_1 : \Pi(x : A).\, P\{\texttt{inl}\ x\} \quad \Gamma \vdash p_2 : \Pi(y : B).\, P\{\texttt{inr}\ y\}}{\Gamma \vdash \texttt{rec}_+\ P\ p_1\ p_2 : \Pi(s : A + B).\, P}$$

$$\texttt{rec}_+\ P\ p_1\ p_2\ (\texttt{inl}\ t_1) \equiv p_1\ t_1$$
$$\texttt{rec}_+\ P\ p_1\ p_2\ (\texttt{inr}\ t_2) \equiv p_2\ t_2$$

Dependent elimination: mind the withdrawal

The other *killer feature* of dependent types.

$$\frac{\Gamma \vdash t_1 : A}{\Gamma \vdash \texttt{inl}\ t_1 : A + B} \qquad \frac{\Gamma \vdash t_2 : B}{\Gamma \vdash \texttt{inr}\ t_2 : A + B}$$

$$\frac{\Gamma, s : A + B \vdash P : \texttt{Type} \quad \Gamma \vdash p_1 : \Pi(x : A).\, P\{\texttt{inl}\ x\} \quad \Gamma \vdash p_2 : \Pi(y : B).\, P\{\texttt{inr}\ y\}}{\Gamma \vdash \texttt{rec}_+\ P\ p_1\ p_2 : \Pi(s : A + B).\, P}$$

$$\texttt{rec}_+\ P\ p_1\ p_2\ (\texttt{inl}\ t_1) \equiv p_1\ t_1$$
$$\texttt{rec}_+\ P\ p_1\ p_2\ (\texttt{inr}\ t_2) \equiv p_2\ t_2$$

Just a beefed up pattern-matching!

$$\texttt{rec}_+\ P\ p_1\ p_2\ s \equiv \texttt{match}\ s\ \texttt{with}\ \texttt{inl}\ x \Rightarrow p_1\ x \mid \texttt{inr}\ y \Rightarrow p_2\ y$$

The type of a branch can change depending on it.

Inductive types

Logically, dependent elimination corresponds to induction principles.

Inductive types

Logically, dependent elimination corresponds to induction principles.

Can even be generalized to fancy types, e.g. equality.

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \texttt{eq}\ A\ t\ u : \texttt{Type}}$$

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma \vdash t : A}{\Gamma \vdash \texttt{refl}\ A\ t : \texttt{eq}\ A\ t\ t}$$

$$\frac{\Gamma, y : A, e : \texttt{eq}\ A\ t\ y \vdash P : \texttt{Type} \qquad \Gamma \vdash p : P\{t, \texttt{refl}\ A\ t\}}{\Gamma \vdash \texttt{rec}_{\texttt{eq}}\ P\ p : \Pi(y : A)\,(e : \texttt{eq}\ A\ t\ y).\, P}$$

$$\texttt{rec}_{\texttt{eq}}\ P\ p\ t\ (\texttt{refl}\ A\ t) \equiv p$$

This implements everything expected from equality!

Wololo

Conversion, the formal way to express that types depend on computation.

$$\frac{\Gamma \vdash M : B \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash M : A}$$

Dallas, ton univers impitoyable

Without entering details too much, types are *also* terms.

$$\Pi(A : \texttt{Type}).\, P \quad \sim \quad \texttt{'a. P} \text{ in OCaml}$$

Dallas, ton univers impitoyable

Without entering details too much, types are *also* terms.

$$\Pi(A : \texttt{Type}).\, P \quad \sim \quad \texttt{'a.} \quad \texttt{P} \quad \text{in OCaml}$$

For instance, we wrote:

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma, x : A \vdash B : \texttt{Type}}{\Gamma \vdash \Pi x : A.\, B : \texttt{Type}}$$

Dallas, ton univers impitoyable

Without entering details too much, types are *also* terms.

$$\Pi(A : \texttt{Type}).\, P \quad \sim \quad \texttt{'a.}\ \ \texttt{P} \ \text{ in OCaml}$$

For instance, we wrote:

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma, x : A \vdash B : \texttt{Type}}{\Gamma \vdash \Pi x : A.\, B : \texttt{Type}}$$

In particular, $\texttt{Type} : \texttt{Type}$.

(Almost.)

Dallas, ton univers impitoyable

Without entering details too much, types are *also* terms.

$$\Pi(A : \texttt{Type}).\, P \quad \sim \quad \texttt{'a. P} \text{ in OCaml}$$

For instance, we wrote:

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma, x : A \vdash B : \texttt{Type}}{\Gamma \vdash \Pi x : A.\, B : \texttt{Type}}$$

In particular, $\texttt{Type} : \texttt{Type}$.

(Almost.)

(For conciseness we will often write $\square := \texttt{Type}$.)

Summary

We can think of CIC as follows.

- A Haskell-like functional language
- $+$ Π generalizing arrows
- $+$ ADTs with generalized pattern-matching
- $+$ inclusion of types in terms
- Proof and computation living in harmony

Summary

We can think of CIC as follows.

- A Haskell-like functional language
- $+\ \Pi$ generalizing arrows
- $+$ ADTs with generalized pattern-matching
- $+$ inclusion of types in terms
- Proof and computation living in harmony

"The earthly paradise of the proof-program correspondence."

“The earthly paradise”?

“The earthly paradise”?

Moi, jamais je pipeaute.

The CIC brothers

Actually not quite one single theory.

Several flags tweaking the kernel:

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

The CIC brothers

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

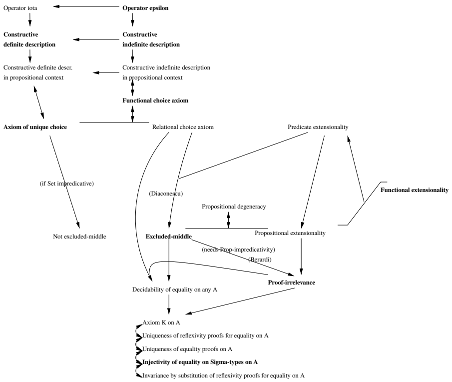
A crazy amount of axioms used in the wild!

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

In the Axiom Jungle

A crazy amount of axioms used in the wild!

The *claſſical ſet-theory* pole:

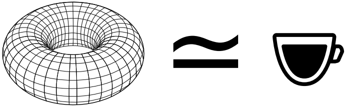
- Excluded middle, UIP, choice

The EXTENSIONAL pole:

- Funext, Propext, Bisim-is-eq

The univalent pole:

- Univalence, what else?



« 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.

□

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.

□

Mathematicians may not care too much, but...

The Most Important Issue of Them All

CIC suffers from an even more **fundamental** flaw.

The Most Important Issue of Them All

CIC suffers from an even more **fundamental** flaw.

- You want to show the wonders of Coq to a fellow programmer
- You fire your favourite IDE
- ... and you're asked the DREADFUL question.

The Most Important Issue of Them All

CIC suffers from an even more **fundamental** flaw.

- You want to show the wonders of Coq to a fellow programmer
- You fire your favourite IDE
- … and you're asked the DREADFUL question.

COULD YOU WRITE A HELLO WORLD PROGRAM PLEASE?

Sad reality

Intuitionistic Logic $\Leftrightarrow$ **Functional** Programming

Sad reality

Intuitionistic Logic $\Leftrightarrow$ **Functional** Programming

Coq is even purer than Haskell:

- No mutable state (obviously)
- No exceptions (Haskell has them somehow)
- No arbitrary recursion
- and also no HELLO WORLD !

Sad reality

Intuitionistic Logic $\Leftrightarrow$ Functional Programming

Coq is even purer than Haskell:

- No mutable state (obviously)
- No exceptions (Haskell has them somehow)
- No arbitrary recursion
- and also no HELLO WORLD !

We want a type theory with effects !

The Good News

Intuitionistic Logic $\Leftrightarrow$ **Functional** Programming

The Good News

Intuitionistic Logic $\Leftrightarrow$ **Functional** Programming

is by folklore the same as

Non-Intuitionistic Logic $\Leftrightarrow$ **Impure** Programming

The Good News

Intuitionistic Logic $\Leftrightarrow$ **Functional** Programming

is by folklore the same as

Non-Intuitionistic Logic $\Leftrightarrow$ **Impure** Programming

- `callcc` gives classical logic
- Delimited continuations prove Markov's principle
- Exceptions implement Markov's rule
- (Certain) presheaves provide univalence
- ...

We want a type theory with **effects** !

Thesis

We want a type theory with effects!

Thesis

We want a type theory with effects!

1. To program more (exceptions, non-termination...)
2. To prove more (classical logic, univalence...)
3. To write Hello World.

Thesis

We want a type theory with effects!

1. To program more (exceptions, non-termination...)
2. To prove more (classical logic, univalence...)
3. To write Hello World.

It's not just randomly coming up with typing rules though.

Thesis

We want a type theory with effects!

1. To program more (exceptions, non-termination...)
2. To prove more (classical logic, univalence...)
3. To write Hello World.

It's not just randomly coming up with typing rules though.

We want a model of type theory with effects.

1. The theory ought to be logically consistent
2. It should be implementable (e.g. decidable type-checking)
3. Other nice properties like canonicity ($\vdash n : \mathbb{N}$ implies $n \rightsquigarrow \mathtt{S} \ldots \mathtt{S}\,\mathtt{0}$)

For the remainder of this talk, we will concentrate on the notion of

models.

For the remainder of this talk, we will concentrate on the notion of

models.

Effectful models of CIC will be discussed on Friday.

(Diabolical laughter.)

The Ancestral Path

Let us assume we defined formally what a model was.

The Ancestral Path

Let us assume we defined formally what a model was.

Assuming we have a model, we can **implement** a new type theory.

Examples: Cubical, F*...

The Ancestral Path

Let us assume we defined formally what a model was.

Assuming we have a model, we can **implement** a new type theory.

Examples: Cubical, F*...

Pro

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

The Ancestral Path

Let us assume we defined formally what a model was.

Assuming we have a model, we can **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. »

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 way out of the conundrum.

One implementation to rule them all...

Teasing

In this talk, I'd like to advocate for a way out of the conundrum.

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

Teasing

In this talk, I'd like to advocate for a way out of the conundrum.

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

One *backend* implementation to rule them all!

In this talk, I'd like to advocate for a way out of the conundrum.

~~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 type theory.

The Set-Theoretical Model

Because Sets are a type theory.

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

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

The Set-Theoretical Model

Because Sets are a type theory.

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

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

Pro

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

The Set-Theoretical Model

Because Sets are a type theory.

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

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

Pro

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

Con

- Forego syntax, computation and decidability
- No effects in sight.
- 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 $[\![A]\!]$
- $\vdash_{\mathsf{CIC}} M : A \quad \Rightarrow \quad [M] \in [\![A]\!]$

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

The Realizability Models

Construct programs that respect properties.

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

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

Pro

- Some preservation of syntax and computability

The Realizability Models

Construct programs that respect properties.

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

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

Pro

- Some preservation of syntax and computability

Con

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

The Categorical Models

Abstract description of type theory.

The Categorical Models

Abstract description of type theory.

Rephrase the rules of CIC in a categorical way.

The Categorical Models

Abstract 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 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 $\mathbf{Set}$:

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

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.

Five seconds of thorough thinking for the sleepy ones.

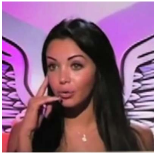
What is a model?

- 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.

Five seconds of thorough thinking for the sleepy ones.

« Non mais allô quoi, this is a *compiler*... »

You are a logician and you don't know 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 $\Rightarrow$ $\neg$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.

Syntactic Models

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 CIC $\subseteq \mathcal{T}$.

Syntactic Models II

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

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

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

Syntactic Models II

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

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

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

Step 2: Flip views and actually pose

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

Syntactic Models II

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

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

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

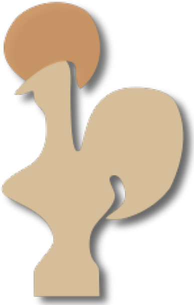
Step 2: Flip views and actually pose

$$\vdash_{\mathcal{T}} M : A \qquad \stackrel{\Delta}{=} \qquad \vdash_{\mathsf{CIC}} [M] : [\![A]\!]$$

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

Anatomy of a syntactic model

COMPILATION



$\vdash_{\mathsf{CIC}^{++}} M : A \qquad \rightsquigarrow \qquad \vdash_{\mathsf{CIC}} [M] : [\![A]\!]$

« 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...

A Few Examples

In the remainder, I'll focus on simple examples of syntactic models.

- Mostly pedagogical
- In particular, no effects involved (?)
- Still funny to mess with CIC

A Few Examples

In the remainder, I'll focus on simple examples of syntactic models.

- Mostly pedagogical
- In particular, no effects involved (?)
- Still funny to mess with CIC

Wait for Friday for me to talk about an effectful CIC.

Models of the day

1. Intensional functions
2. Intensional types, and their utmost horrifying realization
3. Parametricity

Intensional Functions

(A very simple introductory example.)

On Extensionality

Ever heard about *function extensionality*?

$$\texttt{funext} : \Pi(A : \texttt{Type})\,(B : A \to \texttt{Type})(f\,g : \Pi(x : A).\,B\;x).$$
$$(\Pi(x : A).\,f\,x = g\;x) \to f = g$$

On Extensionality

Ever heard about *function extensionality*?

$$\texttt{funext} : \Pi(A : \texttt{Type})\,(B : A \to \texttt{Type})(f\ g : \Pi(x : A).\,B\ x).$$
$$(\Pi(x : A).\,f\ x = g\ x) \to f = g$$

Turns out it is not provable in CIC.

On Extensionality

Ever heard about *function extensionality*?

$$\texttt{funext} : \Pi(A : \texttt{Type})\,(B : A \to \texttt{Type})(f\ g : \Pi(x : A).\, B\ x).$$
$$(\Pi(x : A).\, f\ x = g\ x) \to f = g$$

Turns out it is not provable in CIC.

... even though most sane people add it as an **axiom**.

Negating Functional Extensionality

If it is not provable, let's break it!

Negating Functional Extensionality

If it is not provable, let's break it!

The only thing you know about functions:

$$(\lambda x : A.\, M)\, N \equiv M\{x := N\}$$

Negating Functional Extensionality

If it is not provable, let's break it!

The only thing you know about functions:

$$(\lambda x : A.\, M)\, N \equiv M\{x := N\}$$

Let's take advantage of this by mangling functions in our model.

- Namely, attach a boolean to them
- Will not affect β-reduction
- Will be observable by intensional equality

Technical details

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

(Inductive types mostly untouched.)

Technical details

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

(Inductive types mostly untouched.)

Soundness

We have $\Gamma \vdash M : A$ implies $[\![\Gamma]\!] \vdash [M] : [\![A]\!]$.

Negating Functional Extensionality II

This means that you can extend the source theory with

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda' x : A.\, M : \Pi x : A.\, B}$$

defined as:

$$[\lambda' x : A.\, M] := (\lambda x : [\![A]\!].\, [M], \texttt{false})$$

Rembember:

$$\begin{array}{lcl} [\lambda x : A.\, M] & := & (\lambda x : [\![A]\!].\, [M], \texttt{true}) \\ [M\, N] & := & [M].\pi_1\, [N] \end{array}$$

Negating Functional Extensionality II

This means that you can extend the source theory with

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda' x : A.\, M : \Pi x : A.\, B}$$

defined as:

$$[\lambda' x : A.\, M] := (\lambda x : [\![A]\!].\, [M], \texttt{false})$$

Rembember:

$$\begin{array}{lcl} [\lambda x : A.\, M] & := & (\lambda x : [\![A]\!].\, [M], \texttt{true}) \\ [M\, N] & := & [M].\pi_1\, [N] \end{array}$$

Clearly this new abstraction has the same behaviour as the original one.

$$[(\lambda' x : A.\, M)\, N] \equiv [M\{x := N\}]$$

Negating Functional Extensionality III

Now, it is easy to see how to negate functional extensionality. Consider:

$$\Sigma(f\,g : \mathbb{1} \to \mathbb{1}).\,(\Pi i : \mathbb{1}.\, f\ i = g\ i) \land f \neq g$$

Negating Functional Extensionality III

Now, it is easy to see how to negate functional extensionality. Consider:

$$\Sigma(f\,g : \mathbb{1} \to \mathbb{1}).\,(\Pi i : \mathbb{1}.\, f\ i = g\ i) \wedge f \neq g$$

This is translated into something that is essentially:

$$\Sigma(f\,g : (\mathbb{1} \to \mathbb{1}) \times \texttt{bool}).\,(\Pi i : \mathbb{1}.\, f.\pi_1\ \ i = g.\pi_1\ \ i) \wedge f \neq g$$

(The actual translation is a little noisier, but this does not change the idea.)

Negating Functional Extensionality III

Now, it is easy to see how to negate functional extensionality. Consider:

$$\Sigma(f\,g : \mathbb{1} \to \mathbb{1}).\,(\Pi i : \mathbb{1}.\, f\ i = g\ i) \wedge f \neq g$$

This is translated into something that is essentially:

$$\Sigma(f\,g : (\mathbb{1} \to \mathbb{1}) \times \texttt{bool}).\,(\Pi i : \mathbb{1}.\, f.\pi_1\ \ i = g.\pi_1\ \ i) \wedge f \neq g$$

(The actual translation is a little noisier, but this does not change the idea.)

Take $f := [\lambda x : \mathbb{1}.\, x]$ and $g := [\lambda' x : \mathbb{1}.\, x]$, and *voilá*!

Where We Cheated

We did not explicit the rules of the source theory.

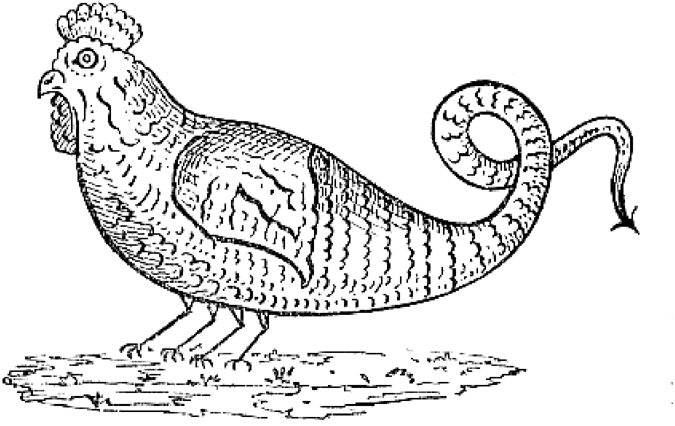
Where We Cheated

We did not explicit the rules of the source theory.

In particular, it is clear that the model invalidates η-rules.

$$\begin{array}{ccc} [\lambda x : A.\, M\ x] & \not\equiv & [M] \\ \equiv & & \equiv \\ (\lambda x : [\![A]\!].\, [M].\pi_1\ x, \mathtt{true}) & \not\equiv & [M] \end{array}$$

It's much harder to negate extensionality while preserving η.
(We'll see than on Friday.)



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 $[M] : [A].\pi_1$

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

Types contain a boolean not used for their inhabitants!

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.

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

- $[\![\mathtt{flip}\ A]\!] \equiv [\![A]\!]$
- 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

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

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

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

Coq is compatible with dynamic types!!!

Parametricity

A Brief Recap on Parametricity

You probably already have heard of **parametricity**.

A Brief Recap on Parametricity

You probably already have heard of **parametricity**.

Originally invented by Reynolds to show properties for System F programs.

Theorems for Free!

A Brief Recap on Parametricity

You probably already have heard of **parametricity**.

Originally invented by Reynolds to show properties for System F programs.

Theorems for Free!

Usually phrased as a set-theoretic relation between terms.

$$\text{If } \vdash_{\mathsf{F}} t : A \text{ then } (t, t) \in [\![A]\!]$$

A Type-Theoretic Parametricity

Instead of System F

What if we used type theory as the **source** theory for parametricity?

A Type-Theoretic Parametricity

Instead of System F

What if we used type theory as the **source** theory for parametricity?

Instead of ZFC

What if we used type theory as the **target** theory for parametricity?

A Type-Theoretic Parametricity

Instead of System F

What if we used type theory as the **source** theory for parametricity?

Instead of ZFC

What if we used type theory as the **target** theory for parametricity?

That's **exactly** what Bernardy-Lasson syntactic translation is about!

A Type-Theoretic Parametricity

Instead of System F

What if we used type theory as the **source** theory for parametricity?

Instead of ZFC

What if we used type theory as the **target** theory for parametricity?

That's **exactly** what Bernardy-Lasson syntactic translation is about!

General idea:

- $A : \square$ is mapped to $[A]_0 : \square$, $[A]_1 : \square$ and $[A]_\varepsilon : [A]_0 \to [A]_1 \to \square$

A Type-Theoretic Parametricity

Instead of System F

What if we used type theory as the **source** theory for parametricity?

Instead of ZFC

What if we used type theory as the **target** theory for parametricity?

That's **exactly** what Bernardy-Lasson syntactic translation is about!

General idea:

- $A : \square$ is mapped to $[A]_0 : \square$, $[A]_1 : \square$ and $[A]_\varepsilon : [A]_0 \to [A]_1 \to \square$
- $M : A$ is mapped to $[M]_i : [A]_i$ and $[M]_\varepsilon : [A]_\varepsilon\ [M]_0\ [M]_1$

A Type-Theoretic Parametricity

Instead of System F

What if we used type theory as the **source** theory for parametricity?

Instead of ZFC

What if we used type theory as the **target** theory for parametricity?

That's **exactly** what Bernardy-Lasson syntactic translation is about!

General idea:

- $A : \square$ is mapped to $[A]_0 : \square$, $[A]_1 : \square$ and $[A]_\varepsilon : [A]_0 \to [A]_1 \to \square$
- $M : A$ is mapped to $[M]_i : [A]_i$ and $[M]_\varepsilon : [A]_\varepsilon\ [M]_0\ [M]_1$
- Variables $x : A$ in the context are triplicated into x_0, x_1, x_ε

The Syntactic Translation

$$[M]_i \quad \equiv \quad M\{\vec{x} := \vec{x}_i\}$$

The Syntactic Translation

$$
\begin{array}{lcl}
[M]_i & \equiv & M\{\vec{x} := \vec{x}_i\} \\
\\
[\Box]_\varepsilon & \equiv & \lambda(A_0 A_1 : \Box).\, A_0 \to A_1 \to \Box \\
[\Pi x : A.\, B]_\varepsilon & \equiv & \lambda(f_0 : [\Pi x : A.\, B]_0)\ (f_1 : [\Pi x : A.\, B]_1). \\
& & \quad \Pi(x_0 : [A]_0)\,(x_1 : [A]_1)\,(x_\varepsilon : [A]_\varepsilon\ x_0\ x_1). \\
& & \qquad [B]_\varepsilon\ (f_0\ x_0)\ (f_1\ x_1)
\end{array}
$$

The Syntactic Translation

$$
\begin{array}{lcl}
[M]_i & \equiv & M\{\vec{x} := \vec{x}_i\} \\
\\
[\square]_\varepsilon & \equiv & \lambda(A_0 A_1 : \square).\, A_0 \to A_1 \to \square \\
[\Pi x : A.\, B]_\varepsilon & \equiv & \lambda(f_0 : [\Pi x : A.\, B]_0)\ (f_1 : [\Pi x : A.\, B]_1). \\
 & & \quad \Pi(x_0 : [A]_0)\,(x_1 : [A]_1)\,(x_\varepsilon : [A]_\varepsilon\ x_0\ x_1). \\
 & & \qquad [B]_\varepsilon\ (f_0\ x_0)\ (f_1\ x_1) \\
[x]_\varepsilon & \equiv & x_\varepsilon \\
[M\ N]_\varepsilon & \equiv & [M]_\varepsilon\ [N]_0\ [N]_1\ [N]_\varepsilon \\
[\lambda x : A.\, M]_\varepsilon & \equiv & \lambda(x_0 : [A]_0)\,(x_1 : [A]_1)\,(x_\varepsilon : [A]_\varepsilon\ x_0\ x_1).\, [M]_\varepsilon
\end{array}
$$

The Syntactic Translation

$$
\begin{array}{lcl}
[M]_i & \equiv & M\{\vec{x} := \vec{x}_i\} \\
\\
[\Box]_\varepsilon & \equiv & \lambda(A_0 A_1 : \Box).\, A_0 \to A_1 \to \Box \\
[\Pi x : A.\, B]_\varepsilon & \equiv & \lambda(f_0 : [\Pi x : A.\, B]_0)\ (f_1 : [\Pi x : A.\, B]_1). \\
 & & \quad \Pi(x_0 : [A]_0)\,(x_1 : [A]_1)\,(x_\varepsilon : [A]_\varepsilon\ x_0\ x_1). \\
 & & \qquad [B]_\varepsilon\ (f_0\ x_0)\ (f_1\ x_1) \\
[x]_\varepsilon & \equiv & x_\varepsilon \\
[M\ N]_\varepsilon & \equiv & [M]_\varepsilon\ [N]_0\ [N]_1\ [N]_\varepsilon \\
[\lambda x : A.\, M]_\varepsilon & \equiv & \lambda(x_0 : [A]_0)\,(x_1 : [A]_1)\,(x_\varepsilon : [A]_\varepsilon\ x_0\ x_1).\, [M]_\varepsilon \\
\\
[\cdot]_\varepsilon & \equiv & \cdot \\
[\Gamma, x : A]_\varepsilon & \equiv & [\Gamma]_\varepsilon, x_0 : [A]_0, x_1 : [A]_1, x_\varepsilon : [A]_\varepsilon\ x_0\ x_1
\end{array}
$$

If $\Gamma \vdash M : A$ then $[\Gamma]_\varepsilon \vdash [M]_i : [A]_i$
$[\Gamma]_\varepsilon \vdash [M]_\varepsilon : [A]_\varepsilon\ [M]_0\ [M]_1.$

Inductive Parametricity

Translation of inductive types is just as simple (but not easy).

- $\mathcal{I} : \square$ is mapped to $\mathcal{I}_\varepsilon : \mathcal{I}_0 \to \mathcal{I}_1 \to \square$
- Constructors are translated pointwise
- Dependent elimination is straightforward

Inductive Parametricity

Translation of inductive types is just as simple (but not easy).

- $\mathcal{I} : \square$ is mapped to $\mathcal{I}_\varepsilon : \mathcal{I}_0 \rightarrow \mathcal{I}_1 \rightarrow \square$
- Constructors are translated pointwise
- Dependent elimination is straightforward

$$
\begin{array}{l}
\texttt{Inductive sum} \\
\quad (A : \square) \\
\quad (B : \square) : \\
\quad \square := \\
\mid \texttt{inl} : \Pi(x : A). \\
\qquad \texttt{sum}\ A\ B \\
\mid \texttt{inr} : \Pi(y : B). \\
\qquad \texttt{sum}\ A\ B
\end{array}
$$

Inductive Parametricity

Translation of inductive types is just as simple (but not easy).

- $\mathcal{I} : \square$ is mapped to $\mathcal{I}_\varepsilon : \mathcal{I}_0 \to \mathcal{I}_1 \to \square$
- Constructors are translated pointwise
- Dependent elimination is straightforward

$$
\begin{array}{l}
\texttt{Inductive sum} \\
\quad (A : \square) \\
\quad (B : \square) : \\
\quad \square := \\
\mid \texttt{inl} : \Pi(x : A). \\
\qquad \texttt{sum}\ A\ B \\
\mid \texttt{inr} : \Pi(y : B). \\
\qquad \texttt{sum}\ A\ B
\end{array}
$$

$$
\begin{array}{l}
\texttt{Inductive sum}_\varepsilon \\
\quad (A_0\, A_1 : \square)\, (A_\varepsilon : A_0 \to A_1 \to \square) \\
\quad (B_0\, B_1 : \square)\, (B_\varepsilon : B_0 \to B_1 \to \square) : \\
\quad \texttt{sum}\ A_0\ B_0 \to \texttt{sum}\ A_1\ B_1 \to \square := \\
\mid \texttt{inl}_\varepsilon : \Pi(x_0 : A_0)\, (x_1 : A_1)\, (x_\varepsilon : A_\varepsilon\ x_0\ x_1). \\
\qquad \texttt{sum}_\varepsilon\ A_0\ A_1\ A_\varepsilon\ B_0\ B_1\ B_\varepsilon\ (\texttt{inl}\ x_0)\ (\texttt{inl}\ x_1) \\
\mid \texttt{inr}_\varepsilon : \Pi(y_0 : B_0)\, (y_1 : B_1)\, (y_\varepsilon : B_\varepsilon\ y_0\ y_1). \\
\qquad \texttt{sum}_\varepsilon\ A_0\ A_1\ A_\varepsilon\ B_0\ B_1\ B_\varepsilon\ (\texttt{inr}\ y_0)\ (\texttt{inr}\ y_1)
\end{array}
$$

Inductive Parametricity

Translation of inductive types is just as simple (but not easy).

- $\mathcal{I} : \square$ is mapped to $\mathcal{I}_\varepsilon : \mathcal{I}_0 \to \mathcal{I}_1 \to \square$
- Constructors are translated pointwise
- Dependent elimination is straightforward

$$
\begin{array}{l}
\texttt{Inductive sum} \\
\quad (A : \square) \\
\quad (B : \square) : \\
\quad \square := \\
\mid \texttt{inl} : \Pi(x : A). \\
\qquad \texttt{sum}\ A\ B \\
\mid \texttt{inr} : \Pi(y : B). \\
\qquad \texttt{sum}\ A\ B
\end{array}
$$

$$
\begin{array}{l}
\texttt{Inductive sum}_\varepsilon \\
\quad (A_0\, A_1 : \square)\,(A_\varepsilon : A_0 \to A_1 \to \square) \\
\quad (B_0\, B_1 : \square)\,(B_\varepsilon : B_0 \to B_1 \to \square) : \\
\quad \texttt{sum}\ A_0\ B_0 \to \texttt{sum}\ A_1\ B_1 \to \square := \\
\mid \texttt{inl}_\varepsilon : \Pi(x_0 : A_0)\,(x_1 : A_1)\,(x_\varepsilon : A_\varepsilon\ x_0\ x_1). \\
\qquad \texttt{sum}_\varepsilon\ A_0\ A_1\ A_\varepsilon\ B_0\ B_1\ B_\varepsilon\ (\texttt{inl}\ x_0)\ (\texttt{inl}\ x_1) \\
\mid \texttt{inr}_\varepsilon : \Pi(y_0 : B_0)\,(y_1 : B_1)\,(y_\varepsilon : B_\varepsilon\ y_0\ y_1). \\
\qquad \texttt{sum}_\varepsilon\ A_0\ A_1\ A_\varepsilon\ B_0\ B_1\ B_\varepsilon\ (\texttt{inr}\ y_0)\ (\texttt{inr}\ y_1)
\end{array}
$$

Parametricity interprets all of CIC!

Parametricity as a Model

Parametricity is as a slightly more complex kind of syntactic model.

Instead of only **one** component, we have **three**.

Parametricity as a Model

Parametricity is as a slightly more complex kind of syntactic model.

Instead of only **one** component, we have **three**.

CIC_p

We define the theory CIC_p as $\Gamma \vdash_{\mathsf{CIC}_p} M : A$ whenever:

- $[\Gamma]_0 \vdash_{\mathsf{CIC}} [M]_0 : [A]_0$
- $[\Gamma]_1 \vdash_{\mathsf{CIC}} [M]_1 : [A]_1$
- $[\Gamma]_\varepsilon \vdash_{\mathsf{CIC}} [M]_\varepsilon : [A]_\varepsilon\ [M]_0\ [M]_1$

Parametricity as a Model

Parametricity is as a slightly more complex kind of syntactic model.

Instead of only **one** component, we have **three**.

CIC_p

We define the theory CIC_p as $\Gamma \vdash_{\mathsf{CIC}_p} M : A$ whenever:

- $[\Gamma]_0 \vdash_{\mathsf{CIC}} [M]_0 : [A]_0$
- $[\Gamma]_1 \vdash_{\mathsf{CIC}} [M]_1 : [A]_1$
- $[\Gamma]_\varepsilon \vdash_{\mathsf{CIC}} [M]_\varepsilon : [A]_\varepsilon\ [M]_0\ [M]_1$

Clearly $\mathsf{CIC} \subseteq \mathsf{CIC}_p$ by soundness.

Parametricity as a Model

Parametricity is as a slightly more complex kind of syntactic model.

Instead of only **one** component, we have **three**.

CIC$_p$

We define the theory CIC$_p$ as $\Gamma \vdash_{\mathsf{CIC}_p} M : A$ whenever:

- $[\Gamma]_0 \vdash_{\mathsf{CIC}} [M]_0 : [A]_0$
- $[\Gamma]_1 \vdash_{\mathsf{CIC}} [M]_1 : [A]_1$
- $[\Gamma]_\varepsilon \vdash_{\mathsf{CIC}} [M]_\varepsilon : [A]_\varepsilon\ [M]_0\ [M]_1$

Clearly CIC $\subseteq$ CIC$_p$ by soundness.

What about the additional expressive power of CIC$_p$?

Patatras!

CIC$_p$ is a **conservative extension** of CIC.

That is, if $\Gamma, M, A \in \mathsf{CIC}$ and $\Gamma \vdash_{\mathsf{CIC}_p} M : A$ then $\Gamma \vdash_{\mathsf{CIC}} M : A$.

Patatras!

CIC$_p$ is a **conservative extension** of CIC.

That is, if $\Gamma, M, A \in \mathsf{CIC}$ and $\Gamma \vdash_{\mathsf{CIC}_p} M : A$ then $\Gamma \vdash_{\mathsf{CIC}} M : A$.

We did not get any expressivity over the CIC common subsystem.

It is trivial to see: just pick the first (or second) projection.

Patatras!

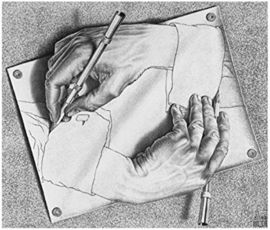
CIC$_p$ is a **conservative extension** of CIC.

That is, if $\Gamma, M, A \in \mathsf{CIC}$ and $\Gamma \vdash_{\mathsf{CIC}_p} M : A$ then $\Gamma \vdash_{\mathsf{CIC}} M : A$.

We did not get any expressivity over the CIC common subsystem.

It is trivial to see: just pick the first (or second) projection.

Parametricity is just repeating properties that were already there.



Echolalia

Is thus CIC_p completely useless?

Echolalia

Is thus CIC_p completely useless?

NO!

Echolalia

Is thus CIC_p completely useless?

NO!

We can still exploit the additional structure to prove independence results!

Echolalia

Is thus CIC_p completely useless?

NO!

We can still exploit the additional structure to prove independence results!

Non-classicality

We have $\not\vdash_{\mathsf{CIC}_p} \Pi(A : \square).\, A + \neg A$.

Corollary

We have $\not\vdash_{\mathsf{CIC}} \Pi(A : \square).\, A + \neg A$.

The Power of Computers

It's extremely tedious to write parametricity proofs by hand.

It's extremely tedious to write parametricity proofs by hand.

Thankfully, we said that this model was a compiler!

Just implement it as a Coq plugin!

- This was done by Keller and Lasson
- Allows to systematically generate horrible proof-terms
- Computer science rules

`https://github.com/coq-community/paramcoq`

Midterm Conclusion

- Syntactic models are a handy tool
- Easier to comprehend (I think)
- Even simple, stupid models are interesting
- Implement them for Coq as mere plugins

Midterm Conclusion

- Syntactic models are a handy tool
- Easier to comprehend (I think)
- Even simple, stupid models are interesting
- Implement them for Coq as mere plugins

Be there on Friday to enter the frightening realm of effects.

Scribitur ad narrandum, non ad probandum.

Merci de votre attention.