Taming EFFECTS in a Dependent World

Pierre-Marie Pédrot

Max Planck Institute for Software Systems

Journées Nationales Géocal-LAC
14th November 2017

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

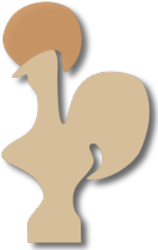
The Pinnacle of the Curry-Howard correspondence

An Effective Object

One implementation to rule them all...

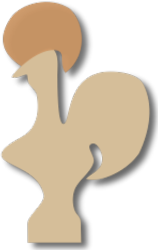
An Effective Object

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 Most Important Issue of Them All

Yet CIC suffers from a **fundamental** flaw.

Yet CIC suffers from a **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

Yet CIC suffers from a **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?

A Well-known Limitation

This is pretty much standard. By the Curry-Howard correspondence

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

A Well-known Limitation

This is pretty much standard. By the Curry-Howard correspondence

Intuitionistic Logic ⇔ **Functional** Programming

That means NO EFFECTS in CIC, amongst which:

- no exceptions, state, non-termination, printing...
- ... and thus no Hello World

Dually, for the same reasons, NO CLASSICAL REASONING.

- Curry-Howard principle: effects extend your logic.

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 $n \rightsquigarrow \mathtt{S} \ldots \mathtt{S}\,\mathtt{0}$)

Aporias

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

Aporias

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

I won't lie: **it is**. But part of this fame is nonetheless due to its models.

Aporias

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

I won't lie: **it is**. But part of this fame is nonetheless due to its models.

Set-theoretical models: because Sets are a (crappy) type theory.

- **Pro:** Sets!
- **Con:** Sets!

Aporias

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

I won't lie: **it is**. But part of this fame is nonetheless due to its models.

Set-theoretical models: because Sets are a (crappy) type theory.

- **Pro:** Sets!
- **Con:** Sets!

Realizability models: construct programs that respect properties.

- **Pro:** Computational, computer-science friendly.
- **Con:** Not foundational (requires an alien meta-theory), not decidable.

Aporias

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

I won't lie: **it is**. But part of this fame is nonetheless due to its models.

Set-theoretical models: because Sets are a (crappy) type theory.

- **Pro:** Sets!
- **Con:** Sets!

Realizability models: construct programs that respect properties.

- **Pro:** Computational, computer-science friendly.
- **Con:** Not foundational (requires an alien meta-theory), not decidable.

Categorical models: abstract description of type theory.

- **Pro:** Abstract, subsumes the two former ones.
- **Con:** Realizability + very low level, gazillion variants, intrisically typed, static.

Curry-Howard Orthodoxy

Instead, let's look at what Curry-Howard provides in simpler settings.

Logical Interpretations $\Leftrightarrow$ Program Translations

Curry-Howard Orthodoxy

Instead, let's look at what Curry-Howard provides in simpler settings.

Logical Interpretations ⇔ Program Translations

On the **programming** side, implement effects using e.g. the *monadic* style.

- A type transformer T, two combinators, a few equations
- Interpret mechanically effectful programs (e.g. in Haskell)

Curry-Howard Orthodoxy

Instead, let's look at what Curry-Howard provides in simpler settings.

Logical Interpretations $\Leftrightarrow$ Program Translations

On the **programming** side, implement effects using e.g. the *monadic* style.

- A type transformer T, two combinators, a few equations
- Interpret mechanically effectful programs (e.g. in Haskell)

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

- 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 $\mathrm{CIC} \subseteq \mathcal{T}$.

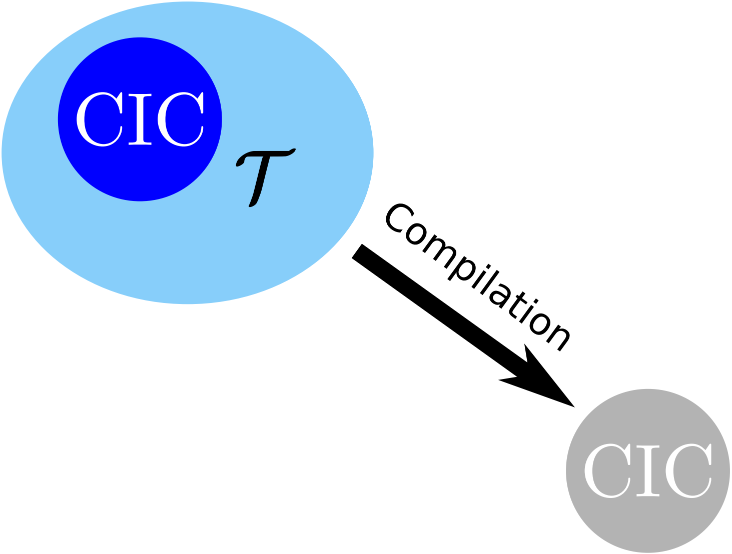
Step 1: Define $[\cdot]$ on the syntax of $\mathcal{T}$ and derive $[\![\cdot]\!]$ from it s.t.

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

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. If you want $\mathrm{CIC} \subseteq \mathcal{T}$,

- The translation must preserve typing (not easy)
- In particular, it must preserve conversion (stay tuned)

Syntactic Models III

Obviously, that's subtle. If you want $\mathrm{CIC} \subseteq \mathcal{T}$,

- The translation must preserve typing (not easy)
- In particular, it must preserve conversion (stay tuned)

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...

Conversion

Dependency entails one major difference with usual program translations.

Conversion

Dependency entails one major difference with usual program translations.

Meet conversion:

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

Conversion

Dependency entails one major difference with usual program translations.

Meet conversion:

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

Bad news 1

Typing rules embed the dynamics of programs!

Conversion

Dependency entails one major difference with usual program translations.

Meet conversion:

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

Bad news 1

Typing rules embed the dynamics of programs!

Combine that with this other observation and we're in trouble.

Bad news 2

Effects make reduction strategies relevant.

A Though Choice

We have two canonical possibilities in presence of effects.

A Though Choice

We have two canonical possibilities in presence of effects.

Call-by-value

- Usual monadic decomposition
- Understandable semantics
- Values still enjoy canonicity
- Good old ML

Call-by-name

- More complex model (CBPV)
- Counter-intuitive behaviours
- Jeopardizes canonicity
- WTF PLT?

Problem I

Recall conversion:

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

Problem I

Recall conversion:

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

In case you forgot your glasses:

CIC has an CBN equational theory.

Problem I

Recall conversion:

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

In case you forgot your glasses:

CIC has an CBN equational theory.

It's unclear what you can do with CBV dependency...

... and probably type terrorists will start crying foul and calling it heresy.

So we have to stick to CBN to please the conservative reviewers.

(But see e.g. comrade Lepigre's agitprop challenging the bourgeois proof theory.)

Problem II

Assuming rightly I don't care about peer pressure, we have another issue.

Problem II

Assuming rightly I don't care about peer pressure, we have another issue.

Monadic encodings don't scale to dependent types.

Problem II

Assuming rightly I don't care about peer pressure, we have another issue.

Monadic encodings don't scale to dependent types.

The reason lies in the typing of bind:

$$\texttt{bind} : T\ A \to (A \to T\ B) \to T\ B.$$

It's seemingly not possible to adapt it to the dependent case!

$$\texttt{dbind} : \Pi(\hat{x} : T\ A).\,(\Pi(x : A).\,T\ (B\ x)) \to T\ (B\ ?).$$

Meanwhile, CBPV naturally extends to dependent types.

We also have to stick to CBN for technical reasons.

Life is Life

Like Homer, we're dragged to the horrible CBN side against our will.

Come on, what could possibly go wronger?

Life is Life

Like Homer, we're dragged to the horrible CBN side against our will.

Come on, what could possibly go wronger?

Dependent elimination + CBN effects $\Rightarrow$ inconsistency.

This is the internal counterpart of the lack of canonicity.

Reduction vs. Effects

- Call-by-name: **functions** well-behaved vs. **inductives** ill-behaved
- Call-by-value: **inductives** well-behaved vs. **functions** ill-behaved

Reduction vs. Effects

- Call-by-name: **functions** well-behaved vs. **inductives** ill-behaved
- Call-by-value: **inductives** well-behaved vs. **functions** ill-behaved

Why is that?

In call-by-name + effects:

$$\begin{array}{ccc} (\lambda x.\, M)\ N \equiv M\{x := N\} & \rightsquigarrow & \text{arbitrary substitution} \\ (\lambda b : \texttt{bool}.\, M)\ \mathbf{fail} & \rightsquigarrow & \text{non-standard booleans} \end{array}$$

In call-by-value + effects:

$$\begin{array}{ccc} (\lambda x.\, M)\ V \equiv M\{x := V\} & \rightsquigarrow & \text{substitute only values} \\ (\lambda b : \texttt{unit}.\, \mathbf{fail}\ b) & \rightsquigarrow & \text{invalid } \eta\text{-rule} \end{array}$$

Eliminating Addiction to Dependence

Recall that dependent elimination is just the induction principle.

For instance, on the boolean type:

$$\frac{\Gamma \vdash M : \mathbb{B} \qquad \Gamma \vdash N_1 : P\{b := \mathtt{true}\} \qquad \Gamma \vdash N_2 : P\{b := \mathtt{false}\}}{\Gamma \vdash \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : P\{b := M\}}$$

This is a statement reflecting canonicity as an internal property in CIC.

Eliminating Addiction to Dependence

Recall that dependent elimination is just the induction principle.

For instance, on the boolean type:

$$\frac{\Gamma \vdash M : \mathbb{B} \qquad \Gamma \vdash N_1 : P\{b := \mathtt{true}\} \qquad \Gamma \vdash N_2 : P\{b := \mathtt{false}\}}{\Gamma \vdash \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 : P\{b := M\}}$$

This is a statement reflecting canonicity as an internal property in CIC.

But there are effectful closed booleans which are neither `true` nor `false`...

Dependent elimination is *hardcore intuitionistic*.

It makes a very strong assumption about the universe of discourse.

Note also that dependent elimination on Σ-types implies AC...

If there is no solution, there is no problem

Dependent elimination + CBN effects $\Rightarrow$ inconsistency.

Two Easy Ways Out!

If there is no solution, there is no problem

Dependent elimination + CBN effects $\Rightarrow$ inconsistency.

Two Easy Ways Out!

1. Embrace inconsistency: truth is a totally overrated social construct.
2. Get into rehab: weaken dependent elimination for a linear fix.

In the remaining of this talk, we will have a look at one instance of each case, namely **exceptions** and **read-only cells**.

Exception

Gotta catch 'em all!

That's *literally* what we are going to do.

The Exceptional Translation

Assume some fixed type of exceptions $\mathbb{E}$.

The exceptional translation extends CIC with

$$\begin{array}{lcll} \texttt{raise}_A & : & \mathbb{E} \rightarrow A & \text{for any } A \\ \texttt{catch}_A & : & A \rightarrow A + \mathbb{E} & \text{for a few specific } A \end{array}$$

satisfying a few expected definitional equations.

The Exceptional Translation

Assume some fixed type of exceptions $\mathbb{E}$.

The exceptional translation extends CIC with

$$\begin{array}{lcll} \mathtt{raise}_A & : & \mathbb{E} \rightarrow A & \text{for any } A \\ \mathtt{catch}_A & : & A \rightarrow A + \mathbb{E} & \text{for a few specific } A \end{array}$$

satisfying a few expected definitional equations.

CBN $\rightsquigarrow$ catching exceptions is limited to positive datatypes (inductive).

In particular, by η-expansion, $\mathtt{raise}_{(\Pi x : A.\, B)}\ e \equiv_\beta \lambda x : A.\, \mathtt{raise}_B\ e$.

The Exceptional Implementation, Negative case

Intuitive idea: translate every $A : \square$ into $[A] : \Sigma A : \square.\, \mathbb{E} \to A$.

$$[\![A]\!] : \square := \pi_1\ [A] \qquad \text{and} \qquad [A]_\varnothing : \mathbb{E} \to [\![A]\!] := \pi_2\ [A]$$

The Exceptional Implementation, Negative case

Intuitive idea: translate every $A : \square$ into $[A] : \Sigma A : \square.\, \mathbb{E} \to A$.

$$[\![A]\!] : \square := \pi_1\ [A] \qquad \text{and} \qquad [A]_\varnothing : \mathbb{E} \to [\![A]\!] := \pi_2\ [A]$$

Because CBN, trivial on the negative fragment:

$$\begin{array}{lcl}
[\![\Pi x : A.\, B]\!] & \equiv & \Pi x : [\![A]\!].\, [\![B]\!] \\
[\Pi x : A.\, B]_\varnothing\ e & \equiv & \lambda x : [\![A]\!].\, [B]_\varnothing\ e \\
[x] & \equiv & x \\
[M\ N] & \equiv & [M]\ [N] \\
[\lambda x : A.\, M] & \equiv & \lambda x : [\![A]\!].\, [M]
\end{array}$$

The Exceptional Implementation, Positive case

The really interesting case is the inductive part of CIC.

How to implement e.g. $[\mathbb{B}]_\varnothing : \mathbb{E} \to [\![\mathbb{B}]\!]$? Or worse $[\bot]_\varnothing : \mathbb{E} \to [\![\bot]\!]$?

The Exceptional Implementation, Positive case

The really interesting case is the inductive part of CIC.

How to implement e.g. $[\mathbb{B}]_\varnothing : \mathbb{E} \to [\![\mathbb{B}]\!]$? Or worse $[\bot]_\varnothing : \mathbb{E} \to [\![\bot]\!]$?

Very simple: add a default case to every inductive type!

$$\texttt{Inductive}\ [\![\mathbb{B}]\!] := [\texttt{true}] : [\![\mathbb{B}]\!] \mid [\texttt{false}] : [\![\mathbb{B}]\!] \mid \mathbb{B}_\varnothing : \mathbb{E} \to [\![\mathbb{B}]\!]$$

The Exceptional Implementation, Positive case

The really interesting case is the inductive part of CIC.

How to implement e.g. $[\mathbb{B}]_\varnothing : \mathbb{E} \to [\![\mathbb{B}]\!]$? Or worse $[\bot]_\varnothing : \mathbb{E} \to [\![\bot]\!]$?

Very simple: add a default case to every inductive type!

$$\texttt{Inductive}\ [\![\mathbb{B}]\!]\ := [\texttt{true}] : [\![\mathbb{B}]\!]\ \mid [\texttt{false}] : [\![\mathbb{B}]\!]\ \mid \mathbb{B}_\varnothing : \mathbb{E} \to [\![\mathbb{B}]\!]$$

Pattern-matching is translated pointwise, except for the new case.

$$\begin{array}{rl} & [\![\Pi P : \mathbb{B} \to \square.\, P\ \texttt{true} \to P\ \texttt{false} \to \Pi b : \mathbb{B}.\, P\ b]\!] \\ \cong & \Pi P : [\![\mathbb{B}]\!] \to [\![\square]\!].\, P\ [\texttt{true}] \to P\ [\texttt{false}] \to \Pi b : [\![\mathbb{B}]\!].\, P\ b \end{array}$$

- If b is $[\texttt{true}]$, use first hypothesis
- If b is $[\texttt{false}]$, use second hypothesis
- If b is an error $\mathbb{B}_\varnothing\ e$, **reraise** e using $[P\ b]_\varnothing\ e$

Time to complain

This gives a syntactic model of all CIC.

Time to complain

This gives a syntactic model of all CIC.

Every type is inhabited by $[\cdot]_{\varnothing}$ and thus the theory is inconsistent!

Time to complain

This gives a syntactic model of all CIC.

Every type is inhabited by $[\cdot]_{\varnothing}$ and thus the theory is inconsistent!

Still usable for programming. Do you whine about OCaml's exceptions?

Plus you can use the target CIC to reason on your effectful programs.

Time to complain

This gives a syntactic model of all CIC.

Every type is inhabited by $[\cdot]_{\varnothing}$ and thus the theory is inconsistent!

Still usable for programming. Do you whine about OCaml's exceptions?

Plus you can use the target CIC to reason on your effectful programs.

Further interest: classical proof extraction. Indeed:

$$[\![\neg\neg A]\!] \cong ([\![A]\!] \to \mathbb{E}) \to \mathbb{E}$$

Allows to prove the following CIC equivalent of Friedman's trick.

Conservativity of classical reasoning on Π^2_0 formulae in CIC

If P and Q are first-order types, $\vdash_{\mathrm{CIC}} \Pi p : P.\, \neg\neg Q$ implies $\vdash_{\mathrm{CIC}} \Pi p : P.\, Q$.

Recovering Consistency

Actually, one can use Bernardy-Lasson parametricity to recover consistency.

Intuition: in addition to $[M] : [\![A]\!]$, produce $[M]_\varepsilon : [\![A]\!]_\varepsilon\ [M]$ where $[\![A]\!]_\varepsilon$ encodes the fact that $[M]$ does not generate uncaught exceptions, e.g.

$$[\![\Pi x : A.\, B]\!]_\varepsilon\ f \quad \equiv \quad \Pi x : [\![A]\!].\, [\![A]\!]_\varepsilon\ x \to [\![B]\!]_\varepsilon\ (f\ x)$$

Recovering Consistency

Actually, one can use Bernardy-Lasson parametricity to recover consistency.

Intuition: in addition to $[M] : [\![A]\!]$, produce $[M]_\varepsilon : [\![A]\!]_\varepsilon\ [M]$ where $[\![A]\!]_\varepsilon$ encodes the fact that $[M]$ does not generate uncaught exceptions, e.g.

$$[\![\Pi x : A.\, B]\!]_\varepsilon\ f \quad \equiv \quad \Pi x : [\![A]\!].\, [\![A]\!]_\varepsilon\ x \rightarrow [\![B]\!]_\varepsilon\ (f\ x)$$

But you still have the right to use exceptions locally!

This is exactly Kreisel's realizability for CIC.

Recovering Consistency

Actually, one can use Bernardy-Lasson parametricity to recover consistency.

Intuition: in addition to $[M] : [\![A]\!]$, produce $[M]_\varepsilon : [\![A]\!]_\varepsilon\ [M]$ where $[\![A]\!]_\varepsilon$ encodes the fact that $[M]$ does not generate uncaught exceptions, e.g.

$$[\![\Pi x : A.\, B]\!]_\varepsilon\ f \quad \equiv \quad \Pi x : [\![A]\!].\, [\![A]\!]_\varepsilon\ x \to [\![B]\!]_\varepsilon\ (f\ x)$$

But you still have the right to use exceptions locally!

This is exactly Kreisel's realizability for CIC.

There is a syntactic model of CIC that proves independence of premise (IP):

$$\Pi(A : \square)\,(P : \mathbb{N} \to \square).\,(\neg A \to \Sigma n : \mathbb{N}.\, P\ n) \to \Sigma n : \mathbb{N}.\, \neg A \to P\ n$$

which is consistent, enjoys canonicity and has decidable type-checking.

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

The Reader Translation

Assume some fixed cell type $\mathbb{R}$.

The reader translation extends type theory with

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

satisfying a few expected definitional equations.

The Reader Translation

Assume some fixed cell type $\mathbb{R}$.

The reader translation extends type theory with

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

satisfying a few expected definitional equations.

The `into` function has unfoldings on type formers:

$$\begin{array}{lcll} \texttt{into}\ (\Pi x : A.\, B)\ r & \equiv & \Pi x : A.\, \texttt{into}\ B\ r & \\ \texttt{into}\ A\ r & \equiv & A & \text{for positive } A \end{array}$$

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 $[M]_r : [A]_r$

On the other side of the CBPV adjunction:

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

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

The Reader Implementation: Inductive Types

PLT tells us we have to take $[\mathbb{B}]_r \equiv \mathbb{B}$.

- It's possible to implement **non-dependent** pattern matching as usual.
- Preserves definitional computation rules

The Reader Implementation: Inductive Types

PLT tells us we have to take $[\mathbb{B}]_r \equiv \mathbb{B}$.

- It's possible to implement **non-dependent** pattern matching as usual.
- Preserves definitional computation rules

But it's **not possible** to implement **dependent** pattern matching!

$$
\begin{aligned}
& [\![\Pi P : \mathbb{B} \to \square.\, P\ \texttt{true} \to P\ \texttt{false} \to \Pi b : \mathbb{B}.\, P\ b]\!]_r \\
\equiv\ & \Pi P : \mathbb{R} \to (\mathbb{R} \to \mathbb{B}) \to \square. \\
& \quad (\Pi s : \mathbb{R}.\, P\ s\ (\lambda\, \_ : \mathbb{R}.\, \texttt{true})) \to (\Pi s : \mathbb{R}.\, P\ s\ (\lambda\, \_ : \mathbb{R}.\, \texttt{false})) \to \\
& \quad\quad \Pi b : \mathbb{R} \to \mathbb{B}.\, P\ r\ b
\end{aligned}
$$

P only holds for two specific values but $b : \mathbb{R} \to \mathbb{B}$ can be anything!

We cannot even test in general that b is extensionally one of those values.

Not All Predicates are Equal

For certain predicates $P : \mathbb{R} \to (\mathbb{R} \to \mathbb{B}) \to \square$, induction still valid though.

Not All Predicates are Equal

For certain predicates $P : \mathbb{R} \to (\mathbb{R} \to \mathbb{B}) \to \square$, induction still valid though.

Indeed, if $P\ r\ b \equiv \Phi\ r\ (b\ r)$ for some Φ, the induction principle becomes

$$(\Pi s : \mathbb{R}.\, \Phi\ s\ \texttt{true}) \to (\Pi s : \mathbb{R}.\, \Phi\ s\ \texttt{false}) \to \Pi b : \mathbb{R} \to \mathbb{B}.\, \Phi\ r\ (b\ r)$$

which is provable by case-analysis on $b\ r$.

Not All Predicates are Equal

For certain predicates $P : \mathbb{R} \to (\mathbb{R} \to \mathbb{B}) \to \square$, induction still valid though.

Indeed, if $P\ r\ b \equiv \Phi\ r\ (b\ r)$ for some Φ, the induction principle becomes

$$(\Pi s : \mathbb{R}.\, \Phi\ s\ \mathtt{true}) \to (\Pi s : \mathbb{R}.\, \Phi\ s\ \mathtt{false}) \to \Pi b : \mathbb{R} \to \mathbb{B}.\, \Phi\ r\ (b\ r)$$

which is provable by case-analysis on $b\ r$.

Such predicates evaluate « immediately » their argument b.

They only rely on the resulting **value**!

This property is completely **independent** from the reader effect.

Moi, j'ai dit linéaire, linéaire ? Comme c'est étrange...

Actually we have a generic **semantic** criterion for valid predicates.

Actually we have a generic **semantic** criterion for valid predicates.

LINEARITY.

- Courtesy of G. Munch, rephrased recently by P. Levy.
- Little to do with « linear use of variables »
- Although tightly linked to linear logic

Linearity in a Nutshell

Defined as an (undecidable) equational property of CBN functions.

A function $f : A \to B$ is linear in A if for all $\hat{x} : \texttt{box}\ A$,

$$f\,(\texttt{match}\ \hat{x}\ \texttt{with}\ \texttt{Box}\ x \Rightarrow x) \equiv \texttt{match}\ \hat{x}\ \texttt{with}\ \texttt{Box}\ x \Rightarrow f\,x$$

where

$$\texttt{Inductive}\ \texttt{box}\ A := \texttt{Box} : A \to \texttt{box}\ A.$$

Linearity in a Nutshell

Defined as an (undecidable) equational property of CBN functions.

A function $f : A \to B$ is linear in A if for all $\hat{x} : \texttt{box}\ A$,

$$f\,(\texttt{match}\ \hat{x}\ \texttt{with}\ \texttt{Box}\ x \Rightarrow x) \equiv \texttt{match}\ \hat{x}\ \texttt{with}\ \texttt{Box}\ x \Rightarrow f\,x$$

where

$$\texttt{Inductive box}\ A := \texttt{Box} : A \to \texttt{box}\ A.$$

- A CBN $f : A \to B$ is linear in A if semantically CBV in A.
- Categorically, f linear iff it is an algebra morphism.
- In a pure language, all functions are linear!

Linear Dependence is All You Need

We restrict dependent elimination in the following way:

$$\frac{\Gamma \vdash M : \mathbb{B} \qquad \ldots \qquad P \text{ linear in } b}{\Gamma \vdash \texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2 : P\{b := M\}}$$

- Can be underapproximated by a **syntactic** criterion
- A new kind of guard condition in CIC
- The CBN doppelgänger of the dreaded **value restriction** in CBV!
- Every predicate can be freely made linear thanks to storage operators

A Bishop-style Type Theory

We can generalize this restriction to form **Baclofen Type Theory**.

- Strict subset of CIC
- Works with our **forcing translation** (LICS 2016)
- Works with our **weaning translation** (LICS 2017)
- Prevents Herbelin's paradox: CIC + callcc inconsistent

A Bishop-style Type Theory

We can generalize this restriction to form **Baclofen Type Theory**.

- Strict subset of CIC
- Works with our **forcing translation** (LICS 2016)
- Works with our **weaning translation** (LICS 2017)
- Prevents Herbelin's paradox: CIC + callcc inconsistent

BTT is the generic theory to deal with dependent effects
« Bishop-style, effect-agnostic type theory »

(Take that, Brouwerian HoTT!)

Implementations

Thanks to the fact we build syntactic models, we can implement them in Coq through a plugin.

`https://github.com/CoqHott/coq-effects`
`https://github.com/CoqHott/exceptional-tt`

- Allows to add effects to Coq just today.
- Implement your favourite effectful operators...
- Compile effectful terms on the fly.
- Allows to reason about them in Coq.

Conclusion

- Effects and dependency: not that complicated if sticking to CBN.
  - But a trade-off about dependent elimination
  - Inconsistency vs. linear dependent elimination
- Even inconsistent theories have practical interest.
  - Exceptions enlarge the dynamic behaviour of your proofs
  - Provide an unsafe hatch that can be used in a safe context
- An experimentally confirmed notion of effectful type theories, BTT
  - Works for forcing, weaning (and `callcc`?)
  - Restriction of dependent elimination on linearity guard condition
  - Conjecture: the correct way to add effects to TT
- Implementation of plugins in Coq: try it out.

Thanks for your attention.