Une Théorie des Types qui fait de l'effet

Pierre-Marie Pédrot

Gallinette (Inria Rennes-à-Nantes)

JFLA 2019

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

Reminder

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

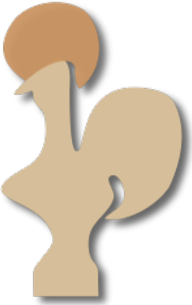
CIC, a very powerful **functional** **programming language**.

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

The Pinnacle of the Curry-Howard correspondence

Syntactic models

COMPILATION



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

« CIC, the LLVM of type theory »

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

We want a type theory with effects !

To `Program` More!

- Obviously you want effects to program
- E.g. state, exceptions, non-termination, continuations...

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

To `Program` More!

- Obviously you want effects to program
- E.g. state, exceptions, non-termination, continuations...

To *Prove* More!

- A well-known fact in the proof theory community
- Curry-Howard $\vdash$ side-effects $\Leftrightarrow$ new axioms
- Archetypical example: `callcc` and classical logic (Griffin, Krivine)

Shameless Propaganda

We already gave two instances of effectful variants of CIC.

Shameless Propaganda

We already gave two instances of effectful variants of CIC.

Forcing (LICS 2016)

- Bread and butter categorical model factory
- *« Forcing: retour de l'être aimé – permis de conduire – désenvoûtement. »*
- Computationally: a glorified monotonous reader monad

Shameless Propaganda

We already gave two instances of effectful variants of CIC.

Forcing (LICS 2016)

- Bread and butter categorical model factory
- *« Forcing: retour de l'être aimé – permis de conduire – désenvoûtement. »*
- Computationally: a glorified monotonous reader monad

Weaning (LICS 2017)

- A generic construction adding effects
- Handles a rather wide class of monads
- Somehow dual to forcing

Shameless Propaganda

We already gave two instances of effectful variants of CIC.

Forcing (LICS 2016)

- Bread and butter categorical model factory
- *« Forcing: retour de l'être aimé – permis de conduire – désenvoûtement. »*
- Computationally: a glorified monotonous reader monad

Weaning (LICS 2017)

- A generic construction adding effects
- Handles a rather wide class of monads
- Somehow dual to forcing

A bit too complex for this introductory course, unfortunately.

Instead

This talk will focus on the big picture.

Instead

This talk will focus on the big picture.

Why did people have so much trouble mixing effects and dependency?

Instead

This talk will focus on the big picture.

Why did people have so much trouble mixing effects and dependency?

Because it's hard.

- Usual models are hard to grasp $\rightsquigarrow$ use syntactic models (done)
- Stuff breaks $\rightsquigarrow$ let's concentrate on that today

Instead

This talk will focus on the big picture.

Why did people have so much trouble mixing effects and dependency?

Because it's hard.

- Usual models are hard to grasp $\rightsquigarrow$ use syntactic models (done)
- Stuff breaks $\rightsquigarrow$ let's concentrate on that today

We might lose part of our type-theoretic soul on the way.

Conversion

Dependency entails one major difference with simpler types.

Conversion

Dependency entails one major difference with simpler types.

Recall conversion:

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

Conversion

Dependency entails one major difference with simpler types.

Recall 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 simpler types.

Recall 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.

You Can't Have Your Cake and Eat It

Effects make reduction strategies relevant.

You Can't Have Your Cake and Eat It

Effects make reduction strategies relevant.

Call-by-value	**Call-by-name**
	

Call-by-value

- ☹ Weaker conversion rule
- ☺ Full dependent elimination
- ☺ Good old ML semantics

Call-by-name

- ☺ Full conversion rule
- ☹ Weaker dependent elimination
- ☹ Strange PL realm

Problems

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

Problems

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

Problem I

CIC has an CBN equational theory.

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

Problems

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

Problem I

CIC has an CBN equational theory.

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

bind	**dbind**
$T\ A \to (A \to T\ B) \to T\ B$	$\Pi(\hat{x} : T\ A).\,(\Pi(x : A).\,T\ (B\ x)) \to T\ (B\ \boxed{\textbf{?}})$

Problems

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

Problem I

CIC has an CBN equational theory.

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

`bind`	`dbind`
$T\ A \rightarrow (A \rightarrow T\ B) \rightarrow T\ B$	$\Pi(\hat{x} : T\ A).\,(\Pi(x : A).\, T\ (B\ x)) \rightarrow T\ (B\ \boxed{\textbf{?}}\,)$

Problem II

CBV monadic encodings don't scale easily to dependent types.

Problems

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

Problem I

CIC has an CBN equational theory.

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

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

Problem II

CBV monadic encodings don't scale easily to dependent types.

We have* to stick to call-by-name!

What can go wronger?

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

What can go wronger?

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

In **call-by-name** + effects:

$$(\lambda x.\, M)\ N \equiv M\{x := N\} \quad \rightsquigarrow \quad \textbf{arbitrary substitution}$$

$$(\lambda b : \texttt{bool}.\, M)\ \mathbf{fail} \quad \rightsquigarrow \quad \textbf{non-standard booleans}$$

In **call-by-value** + effects:

$$(\lambda x.\, M)\ V \equiv M\{x := V\} \quad \rightsquigarrow \quad \textbf{substitute only values}$$

$$(\lambda b : \texttt{unit}.\, \mathbf{fail}\ b) \quad \rightsquigarrow \quad \textbf{invalid } \eta\textbf{-rule}$$

Eliminating Addiction to Dependence

Recall that dependent elimination is just the induction principle.

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

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.

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

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 incompatible with CBN effects.

Eliminating Addiction to Dependence

Recall that dependent elimination is just the induction principle.

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

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 incompatible with CBN effects.

Takeaway

Dependent elimination is *hardcore intuitionistic*.

In This Talk

We will focus on two simple examples of effects

- Part I: Read-only cell
- Part II: Exceptions

In This Talk

We will focus on two simple examples of effects

- Part I: Read-only cell
- Part II: Exceptions

They feature fundamental interactions between effects and dependency.

In This Talk

We will focus on two simple examples of effects

- Part I: Read-only cell
- Part II: Exceptions

They feature fundamental interactions between effects and dependency.

We will implement them with syntactic models.

In call-by-name !

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

Overview

Essentially the same as Haskell's reader effect.

- There is a global unnamed cell
- That can be read
- That can be updated **in a well-scoped way**

Not quite a state!

To add insult to injury, we're in call-by-name.

The Reader Translation

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

The Reader Translation

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

The reader translation extends CIC into $\mathsf{CIC}_{\mathbb{R}}$, with

$$\begin{array}{rrcl} & \texttt{read} & : & \mathbb{R} \\ & \texttt{into} & : & \square \to \mathbb{R} \to \square \\ & \texttt{enter} & : & \Pi(A : \square).\, A \to \Pi r : \mathbb{R}.\, \texttt{into}\ A\ r \\ (\text{morally} & \texttt{enter} & : & \Pi(A : \square).\, A \to \mathbb{R} \to A) \end{array}$$

The Reader Translation

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

The reader translation extends CIC into $\mathrm{CIC}_{\mathbb{R}}$, with

$$
\begin{array}{rrcl}
 & \mathtt{read} & : & \mathbb{R} \\
 & \mathtt{into} & : & \square \rightarrow \mathbb{R} \rightarrow \square \\
 & \mathtt{enter} & : & \Pi(A : \square).\, A \rightarrow \Pi r : \mathbb{R}.\, \mathtt{into}\ A\ r \\
(\text{morally} & \mathtt{enter} & : & \Pi(A : \square).\, A \rightarrow \mathbb{R} \rightarrow A)
\end{array}
$$

satisfying the expected definitional equations, e.g.

$$
\begin{array}{rcl}
\mathtt{enter}\ \mathbb{R}\ \mathtt{read}\ M & \equiv & M \\
\mathtt{enter}\ \mathbb{R}\ M\ \mathtt{read} & \equiv & M
\end{array}
$$

Remember, we're call-by-name...

Enters `into`

The `into` is a mere typing artifact.

$$\texttt{into} : \square \rightarrow \mathbb{R} \rightarrow \square$$

It has unfoldings on type formers:

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

Enters $\texttt{into}$

The $\texttt{into}$ is a mere typing artifact.

$$\texttt{into} : \square \to \mathbb{R} \to \square$$

It has unfoldings on type formers:

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

together with the following conversion:

$$\texttt{into} \equiv \texttt{enter}\ \square$$

$\texttt{into}$ is $\texttt{enter}$, but there is a typing loop.

Recall that:

$$\texttt{enter} : \Pi(A : \square).\, A \to \Pi r : \mathbb{R}.\, \texttt{into}\ A\ r$$

The Reader Implementation

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

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

The Reader Implementation

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

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

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

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

The Reader Implementation

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

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

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

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

Soundness

We have $\Gamma \vdash M : A$ implies $[\![\Gamma]\!], r : \mathbb{R} \vdash [M]_r : [A]_r$.

The Reader Implementation: Inductive Types

PLT tells us we have to take $[A + B]_r \equiv [\![A]\!] + [\![B]\!]$.

$$\begin{array}{rcl} [A + B]_r & \equiv & [\![A]\!] + [\![B]\!] \\ [\mathtt{inl}\ M]_r & \equiv & \mathtt{inl}\ (\Pi s : \mathbb{R}.\, [M]_s) \\ [\mathtt{inr}\ M]_r & \equiv & \mathtt{inr}\ (\Pi s : \mathbb{R}.\, [M]_s) \end{array}$$

The Reader Implementation: Inductive Types

PLT tells us we have to take $[A + B]_r \equiv [\![A]\!] + [\![B]\!]$.

$$\begin{array}{rcl} [A + B]_r & \equiv & [\![A]\!] + [\![B]\!] \\ [\mathtt{inl}\ M]_r & \equiv & \mathtt{inl}\ (\Pi s : \mathbb{R}.\, [M]_s) \\ [\mathtt{inr}\ M]_r & \equiv & \mathtt{inr}\ (\Pi s : \mathbb{R}.\, [M]_s) \end{array}$$

It's possible to implement **non-dependent** pattern-matching as usual.

The Reader Implementation: Inductive Types

PLT tells us we have to take $[A + B]_r \equiv [\![A]\!] + [\![B]\!]$.

$$\begin{array}{rcl} [A + B]_r & \equiv & [\![A]\!] + [\![B]\!] \\ [\mathtt{inl}\ M]_r & \equiv & \mathtt{inl}\ (\Pi s : \mathbb{R}.\, [M]_s) \\ [\mathtt{inr}\ M]_r & \equiv & \mathtt{inr}\ (\Pi s : \mathbb{R}.\, [M]_s) \end{array}$$

It's possible to implement **non-dependent** pattern-matching as usual.

$$[\mathtt{elim}_+]_r : [\![\Pi P : \square.\, (A \to P) \to (B \to P) \to A + B \to P]\!] \equiv \Pi(P : \mathbb{R} \to \square).$$
$$(\Pi s : \mathbb{R}.\, [\![A]\!] \to P\ s) \to (\Pi s : \mathbb{R}.\, [\![B]\!] \to P\ s) \to (\mathbb{R} \to [\![A]\!] + [\![B]\!]) \to P\ r$$

$$\begin{array}{rcl} \mathtt{elim}_+\ P\ N_l\ N_r\ (\mathtt{inl}\ M) & \equiv & N_l\ M \\ \mathtt{elim}_+\ P\ N_l\ N_r\ (\mathtt{inr}\ M) & \equiv & N_r\ M \end{array}$$

Uh-oh

Unfortunately, It's **not possible** to implement **dependent** elimination!

$$[\![\Pi P.\,(\Pi(x : A).\,P\ (\mathtt{inl}\ x)) \rightarrow (\Pi(y : B).\,P\ (\mathtt{inr}\ y)) \rightarrow \Pi b : A + B.\,P\ b]\!]$$

$$\equiv$$

$$\begin{aligned}
&\Pi P : \mathbb{R} \rightarrow (\mathbb{R} \rightarrow [\![A]\!] + [\![B]\!]) \rightarrow \square.\\
&\quad (\Pi(s : \mathbb{R})\,(x : [\![A]\!]).\,P\ s\ (\lambda\,_ : \mathbb{R}.\,\mathtt{inl}\ x)) \rightarrow\\
&\quad (\Pi(s : \mathbb{R})\,(y : [\![B]\!]).\,P\ s\ (\lambda\,_ : \mathbb{R}.\,\mathtt{inr}\ y)) \rightarrow\\
&\qquad \Pi(b : \mathbb{R} \rightarrow [\![A]\!] + [\![B]\!]).\,P\ r\ b
\end{aligned}$$

Uh-oh

Unfortunately, It's **not possible** to implement **dependent** elimination!

$$[\![\Pi P.\, (\Pi(x : A).\, P\ (\mathtt{inl}\ x)) \to (\Pi(y : B).\, P\ (\mathtt{inr}\ y)) \to \Pi b : A + B.\, P\ b]\!]$$

$$\equiv$$

$$\begin{array}{l} \Pi P : \mathbb{R} \to (\mathbb{R} \to [\![A]\!] + [\![B]\!]) \to \square. \\ \quad (\Pi(s : \mathbb{R})\, (x : [\![A]\!]).\, P\ s\ (\lambda\, \_ : \mathbb{R}.\, \mathtt{inl}\ x)) \to \\ \quad (\Pi(s : \mathbb{R})\, (y : [\![B]\!]).\, P\ s\ (\lambda\, \_ : \mathbb{R}.\, \mathtt{inr}\ y)) \to \\ \qquad \Pi(b : \mathbb{R} \to [\![A]\!] + [\![B]\!]).\, P\ r\ b \end{array}$$

P only holds for two constant values but b can be anything!

Uh-oh

Unfortunately, It's **not possible** to implement **dependent** elimination!

$$[\![\Pi P.\,(\Pi(x : A).\,P\;(\texttt{inl}\;x)) \rightarrow (\Pi(y : B).\,P\;(\texttt{inr}\;y)) \rightarrow \Pi b : A + B.\,P\;b]\!]$$

$$\equiv$$

$$\begin{array}{l}
\Pi P : \mathbb{R} \rightarrow (\mathbb{R} \rightarrow [\![A]\!] + [\![B]\!]) \rightarrow \square. \\
\quad (\Pi(s : \mathbb{R})\,(x : [\![A]\!]).\,P\;s\;\boxed{(\lambda\,_ : \mathbb{R}.\,\texttt{inl}\;x)}) \rightarrow \\
\quad (\Pi(s : \mathbb{R})\,(y : [\![B]\!]).\,P\;s\;\boxed{(\lambda\,_ : \mathbb{R}.\,\texttt{inr}\;y)}) \rightarrow \\
\qquad \Pi(b : \mathbb{R} \rightarrow [\![A]\!] + [\![B]\!]).\,P\;r\;\boxed{b}
\end{array}$$

P only holds for two constant values but b can be anything!

Reminder

Dependent elimination is incompatible with CBN effects.

Not All Predicates are Equal

In general through $[\cdot]_r$ predicates have the following type:

$$P : \mathbb{R} \to (\mathbb{R} \to [\![A]\!] + [\![B]\!]) \to \square$$

Not All Predicates are Equal

In general through $[\cdot]_r$ predicates have the following type:

$$P : \mathbb{R} \to (\mathbb{R} \to [\![A]\!] + [\![B]\!]) \to \square$$

Assume there is $\Phi : \mathbb{R} \to [\![A]\!] + [\![B]\!] \to \square$ s.t.

$$P\ r\ b := \Phi\ r\ (b\ r)$$

Not All Predicates are Equal

In general through $[\cdot]_r$ predicates have the following type:

$$P : \mathbb{R} \to (\mathbb{R} \to [\![A]\!] + [\![B]\!]) \to \square$$

Assume there is $\Phi : \mathbb{R} \to [\![A]\!] + [\![B]\!] \to \square$ s.t.

$$P\ r\ b := \Phi\ r\ (b\ r)$$

In this case, induction principle becomes

$$\begin{array}{r}(\Pi(s : \mathbb{R})\,(x : [\![A]\!]).\,\Phi\ s\ (\mathtt{inl}\ x)) \to \\ (\Pi(s : \mathbb{R})\,(y : [\![B]\!]).\,\Phi\ s\ (\mathtt{inr}\ y)) \to \\ \Pi(b : \mathbb{R} \to [\![A]\!] + [\![B]\!]).\,\Phi\ r\ (b\ r)\end{array}$$

This **is** provable!

Not All Predicates are Equal

In general through $[\cdot]_r$ predicates have the following type:

$$P : \mathbb{R} \to (\mathbb{R} \to [\![A]\!] + [\![B]\!]) \to \square$$

Assume there is $\Phi : \mathbb{R} \to [\![A]\!] + [\![B]\!] \to \square$ s.t.

$$P\ r\ b := \Phi\ r\ (b\ r)$$

In this case, induction principle becomes

$$\begin{array}{r}(\Pi(s : \mathbb{R})\,(x : [\![A]\!]).\,\Phi\ s\ (\mathtt{inl}\ x)) \to \\ (\Pi(s : \mathbb{R})\,(y : [\![B]\!]).\,\Phi\ s\ (\mathtt{inr}\ y)) \to \\ \Pi(b : \mathbb{R} \to [\![A]\!] + [\![B]\!]).\,\Phi\ r\ (b\ r)\end{array}$$

This **is** provable!

Induction is still valid for predicates that evaluate **eagerly** their argument.

Freely Turning Eager

Fact 1

There is a whole class of such **eager** predicates.

For instance, if the predicate P starts with a pattern-matching.

$$P \ := \lambda b.\, \texttt{match}\ b\ \texttt{with}\ \texttt{inl}\ x \to u_1 \mid \texttt{inr}\ y \to u_2$$

$$[P]_r := \lambda b.\, \texttt{match}\ b\ r\ \texttt{with}\ \texttt{inl}\ x \to [u_1]_r \mid \texttt{inr}\ y \to [u_2]_r$$

Freely Turning Eager

Fact 1

There is a whole class of such **eager** predicates.

For instance, if the predicate P starts with a pattern-matching.

$$P \ := \lambda b.\,\mathtt{match}\ b\ \mathtt{with}\ \mathtt{inl}\ x \to u_1 \mid \mathtt{inr}\ y \to u_2$$

$$[P]_r := \lambda b.\,\mathtt{match}\ b\ r\ \mathtt{with}\ \mathtt{inl}\ x \to [u_1]_r \mid \mathtt{inr}\ y \to [u_2]_r$$

Fact 2

Any predicate can be turned into an eager predicate.

Thanks to **storage operators**.

Storage Operators

Storage operator are a technique to implement CBV in CBN.

Originally from classical realizability, to implement **induction** (ahem?).

Storage Operators

Storage operator are a technique to implement CBV in CBN.

Originally from classical realizability, to implement **induction** (ahem?).

$$\theta_{A+B} : A + B \to (A + B \to \square) \to \square$$

Storage Operators

Storage operator are a technique to implement CBV in CBN.

Originally from classical realizability, to implement **induction** (ahem?).

$$\theta_{A+B} : A + B \to (A + B \to \square) \to \square$$

- This is a CPS

Storage Operators

Storage operator are a technique to implement CBV in CBN.

Originally from classical realizability, to implement **induction** (ahem?).

$$\theta_{A+B} : A + B \to (A + B \to \square) \to \square$$

- This is a CPS
- It can be implemented using non-dependent elimination!

$$\theta_{A+B}\ b\ P := \texttt{match}\ b\ \texttt{with inl}\ x \Rightarrow P\ (\texttt{inl}\ x) \mid \texttt{inr}\ y \Rightarrow P\ (\texttt{inr}\ y)$$

Storage Operators

Storage operator are a technique to implement CBV in CBN.

Originally from classical realizability, to implement **induction** (ahem?).

$$\theta_{A+B} : A + B \to (A + B \to \square) \to \square$$

- This is a CPS
- It can be implemented using non-dependent elimination!

$$\theta_{A+B}\ b\ P := \mathtt{match}\ b\ \mathtt{with}\ \mathtt{inl}\ x \Rightarrow P\ (\mathtt{inl}\ x)\ |\ \mathtt{inr}\ y \Rightarrow P\ (\mathtt{inr}\ y)$$

- In presence of dependent elimination,

$$\vdash_{\mathsf{CIC}} \theta_{A+B}\ b\ P = P\ b$$

Fixing Elimination

Dependent elimination is **not valid** in general.

$$\not\vdash_{\mathsf{CIC}_{\mathbb{R}}} (\Pi(x : A).\, P\ (\mathtt{inl}\ x)) \rightarrow (\Pi(y : B).\, P\ (\mathtt{inr}\ y)) \rightarrow \Pi b : A + B.\, P\ b$$

Fixing Elimination

Dependent elimination is **not valid** in general.

$$\not\vdash_{\mathsf{CIC}_{\mathbb{R}}} (\Pi(x : A).\, P\ (\texttt{inl}\ x)) \to (\Pi(y : B).\, P\ (\texttt{inr}\ y)) \to \Pi b : A + B.\, P\ b$$

Dependent elimination is **valid** if first stored!.

$$\vdash_{\mathsf{CIC}_{\mathbb{R}}} (\Pi(x : A).\, P\ (\texttt{inl}\ x)) \to (\Pi(y : B).\, P\ (\texttt{inr}\ y)) \to \Pi b : A + B.\, \theta_{A+B}\ b\ P$$

Because θ_{A+B} turns any predicate into an eager one.

Linearity

Induction is still valid for predicates that evaluate **eagerly** their argument.

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

Linearity

Induction is still valid for predicates that evaluate **eagerly** their argument.

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

LINEARITY.

- Little to do with « linear use of variables, but tightly linked to LL
- Defined as an (undecidable) equational property of CBN functions
- A generalization of **strictness**
- 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 **guard condition**
- The CBN doppelgänger of the dreaded **value restriction** in CBV!
- Any predicate can be freely made linear thanks to **storage operators**

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 **guard condition**
- The CBN doppelgänger of the dreaded **value restriction** in CBV!
- Any predicate can be freely made linear thanks to **storage operators**

This restriction forms **Baclofen Type Theory**.

Outrageous claim

BTT is the generic theory to deal with dependent effects

The Exceptional Type Theory

(a.k.a. the Curry-Howard-Shadok correspondence)

An extension of CIC rooted in Shadok wisdom.

“The more it fails, the more likely it will eventually succeed.”

An extension of CIC rooted in Shadok wisdom.

"THE MORE IT FAILS, THE MORE LIKELY IT WILL EVENTUALLY SUCCEED."

- ☺ Add a failure mechanism to CIC
- ☺ Fully computational exceptions
- ☺ Features full conversion
- ☺ Features full dependent elimination

An extension of CIC rooted in Shadok wisdom.

"THE MORE IT FAILS, THE MORE LIKELY IT WILL EVENTUALLY SUCCEED."

- ☺ Add a failure mechanism to CIC
- ☺ Fully computational exceptions
- ☺ Features full conversion
- ☺ Features full dependent elimination
- 😕 Didn't I say this was not possible???

An extension of CIC rooted in Shadok wisdom.

“THE MORE IT FAILS, THE MORE LIKELY IT WILL EVENTUALLY SUCCEED.”

- ☺ Add a failure mechanism to CIC
- ☺ Fully computational exceptions
- ☺ Features full conversion
- ☺ Features full dependent elimination
- 😕 Didn’t I say this was not possible???

The Exceptional Type Theory: Overview

The exceptional type theory extends vanilla CIC with

$$\begin{array}{rcl} \mathbf{E} & : & \square \\ \mathtt{raise} & : & \Pi A : \square.\, \mathbf{E} \to A \end{array}$$

The Exceptional Type Theory: Overview

The exceptional type theory extends vanilla CIC with

$$\begin{array}{rcl} \mathbf{E} & : & \square \\ \texttt{raise} & : & \Pi A : \square.\, \mathbf{E} \rightarrow A \end{array}$$

As hinted before, we need to be call-by-name to feature full conversion.

$$\begin{array}{lcl} \texttt{raise}\ (\Pi x : A.\, B)\ e & \equiv & \lambda x : A.\, \texttt{raise}\ B\ e \\ \texttt{match}\ (\texttt{raise}\ \mathcal{I}\ e)\ \texttt{ret}\ P\ \texttt{with}\ \vec{p} & \equiv & \texttt{raise}\ (P\ (\texttt{raise}\ \mathcal{I}\ e))\ e \end{array}$$

where $P : \mathcal{I} \rightarrow \square$.

The Exceptional Type Theory: Overview

The exceptional type theory extends vanilla CIC with

$$\begin{array}{rcl} \mathbf{E} & : & \square \\ \mathtt{raise} & : & \Pi A : \square.\, \mathbf{E} \rightarrow A \end{array}$$

As hinted before, we need to be call-by-name to feature full conversion.

$$\begin{array}{rcl} \mathtt{raise}\ (\Pi x : A.\, B)\ e & \equiv & \lambda x : A.\, \mathtt{raise}\ B\ e \\ \mathtt{match}\ (\mathtt{raise}\ \mathcal{I}\ e)\ \mathtt{ret}\ P\ \mathtt{with}\ \vec{p} & \equiv & \mathtt{raise}\ (P\ (\mathtt{raise}\ \mathcal{I}\ e))\ e \end{array}$$

where $P : \mathcal{I} \rightarrow \square$.

Remark that in call-by-name, if $M : A \rightarrow B$, in general

$$M\ (\mathtt{raise}\ A\ e) \quad \not\equiv \quad \mathtt{raise}\ B\ e$$

for otherwise we would not have $(\lambda x : A.\, M)\ N \equiv M\{x := N\}$.

Catch Me If You Can

Remember that on functions:

$$\texttt{raise}\ (\Pi x : A.\, B)\ e \quad \equiv \quad \lambda x : A.\, \texttt{raise}\ B\ e$$

It means catching exceptions is limited to positive datatypes!

Catch Me If You Can

Remember that on functions:

$$\texttt{raise}\ (\Pi x : A.\, B)\ e \quad \equiv \quad \lambda x : A.\, \texttt{raise}\ B\ e$$

It means catching exceptions is limited to positive datatypes!

For inductive types, this is a **generalized induction principle**.

$$\begin{array}{ll} \texttt{catch}_{\mathbb{B}} : & \Pi P : \mathbb{B} \to \Box. \\ & \quad P\ \texttt{true} \to \\ & \quad P\ \texttt{false} \to \\ & \quad (\Pi e : \mathbf{E}.\, P\ (\texttt{raise}\ \mathbb{B}\ e)) \to \\ & \Pi b : \mathbb{B}.\, P\ b \end{array} \qquad \begin{array}{ll} \mathbb{B}_{\texttt{rect}} : & \Pi P : \mathbb{B} \to \Box. \\ & \quad P\ \texttt{true} \to \\ & \quad P\ \texttt{false} \to \\ & \\ & \Pi b : \mathbb{B}.\, P\ b \end{array}$$

where

$$\begin{array}{lcl} \texttt{catch}_{\mathbb{B}}\ P\ p_t\ p_f\ p_e\ \texttt{true} & \equiv & p_t \\ \texttt{catch}_{\mathbb{B}}\ P\ p_t\ p_f\ p_e\ \texttt{false} & \equiv & p_f \\ \texttt{catch}_{\mathbb{B}}\ P\ p_t\ p_f\ p_e\ (\texttt{raise}\ \mathbb{B}\ e) & \equiv & p_e\ e \end{array}$$

The Exceptional Implementation

Let's implement the exceptional type theory into CIC!

The Exceptional Implementation

Let's implement the exceptional type theory into CIC!

- Source is a CBN theory, so usual monadic encoding won't work.
- We use a variant of our previous weaning translation.
- All typing and computations rules mentioned before hold for free.

The Exceptional Implementation

Let's implement the exceptional type theory into CIC!

- Source is a CBN theory, so usual monadic encoding won't work.
- We use a variant of our previous weaning translation.
- All typing and computations rules mentioned before hold for free.

Let's call the exceptional type theory $\mathcal{T}_{\mathbb{E}}$ to disambiguate it from CIC.

The Exceptional Implementation

Let's implement the exceptional type theory into CIC!

- Source is a CBN theory, so usual monadic encoding won't work.
- We use a variant of our previous weaning translation.
- All typing and computations rules mentioned before hold for free.

Let's call the exceptional type theory $\mathcal{T}_{\mathbb{E}}$ to disambiguate it from CIC.

Only parameter of the translation: a fixed type of exceptions in the target.

$$\vdash_{\mathsf{CIC}} \mathbb{E} : \square$$

The Exceptional Implementation, Negative case

Intuition: $\vdash_{\mathcal{T}_{\mathbb{E}}} A : \square \qquad \rightsquigarrow \qquad \vdash_{\mathsf{CIC}} [A] : \Sigma A : \square.\, \mathbb{E} \to A.$

Every exceptional type comes with its own implementation of failure!

The Exceptional Implementation, Negative case

Intuition: $\vdash_{\mathcal{T}_{\mathbb{E}}} A : \square \quad \rightsquigarrow \quad \vdash_{\mathsf{CIC}} [A] : \Sigma A : \square.\, \mathbb{E} \to A.$

Every exceptional type comes with its own implementation of failure!

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

$$\begin{array}{lcl} [\![\square]\!] & \equiv & \Sigma A : \square.\, \mathbb{E} \to A \\ [\square]_{\varnothing}\ e & \equiv & \ldots \\ [\![\Pi x : A.\, B]\!] & \equiv & \Pi x : [\![A]\!].\, [\![B]\!] \\ [\Pi x : A.\, B]_{\varnothing}\ e & \equiv & \lambda x : [\![A]\!].\, [B]_{\varnothing}\ e \end{array}$$

The Exceptional Implementation, Negative case

Intuition: $\vdash_{\mathcal{T}_{\mathbb{E}}} A : \square \quad \rightsquigarrow \quad \vdash_{\mathsf{CIC}} [A] : \Sigma A : \square.\, \mathbb{E} \to A.$

Every exceptional type comes with its own implementation of failure!

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

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

The Exceptional Implementation, Negative case

Intuition: $\vdash_{\mathcal{T}_{\mathbb{E}}} A : \square \quad \rightsquigarrow \quad \vdash_{\mathsf{CIC}} [A] : \Sigma A : \square.\, \mathbb{E} \to A.$

Every exceptional type comes with its own implementation of failure!

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

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

If $\Gamma \vdash_{\mathsf{CIC}} M : A$ then $[\![\Gamma]\!] \vdash_{\mathsf{CIC}} [M] : [\![A]\!]$.

The Exceptional Implementation, Failure

It is straightforward to implement the failure operation.

$$\begin{array}{lcl} \mathbf{E} & : & \square \\ \texttt{raise} & : & \Pi A : \square.\, \mathbf{E} \rightarrow A \end{array}$$

The Exceptional Implementation, Failure

It is straightforward to implement the failure operation.

$$\begin{array}{rcl} \mathbf{E} & : & \square \\ \texttt{raise} & : & \Pi A : \square.\, \mathbf{E} \to A \end{array}$$

$$\begin{array}{rcl} [\mathbf{E}] & : & \Sigma A : \square.\, \mathbb{E} \to A \\ [\mathbf{E}] & := & (\mathbb{E}, \lambda e : \mathbb{E}.\, e) \\ \\ [\texttt{raise}] & : & \Pi A_0 : (\Sigma A : \square.\, \mathbb{E} \to A).\, \mathbb{E} \to \pi_1\ A_0 \\ [\texttt{raise}] & := & \pi_2 \end{array}$$

The Exceptional Implementation, Failure

It is straightforward to implement the failure operation.

$$\begin{array}{lcl} \mathbf{E} & : & \square \\ \texttt{raise} & : & \Pi A : \square.\, \mathbf{E} \to A \end{array}$$

$$\begin{array}{lcl} [\mathbf{E}] & : & \Sigma A : \square.\, \mathbb{E} \to A \\ [\mathbf{E}] & := & (\mathbb{E}, \lambda e : \mathbb{E}.\, e) \\ \\ [\texttt{raise}] & : & \Pi A_0 : (\Sigma A : \square.\, \mathbb{E} \to A).\, \mathbb{E} \to \pi_1\ A_0 \\ [\texttt{raise}] & := & \pi_2 \end{array}$$

Computational rules trivially hold!

$$\begin{array}{ccc} [\texttt{raise}\ (\Pi x : A.\, B)\ e] & \equiv & [\lambda x : A.\, \texttt{raise}\ B\ e] \\ ||| & & ||| \\ \pi_2\ ((\Pi x : [\![A]\!].\, [\![B]\!]), (\lambda (e : \mathbb{E})\, (x : [\![A]\!]).\, \pi_2\ [\![B]\!]\ e))\ [e] & \equiv & \lambda x : [\![A]\!].\, \pi_2\ [B]\ [e] \end{array}$$

The Exceptional Implementation, Positive case

The really interesting case is the inductive part of CIC.

How to implement $[\mathbb{B}]_{\varnothing} : \mathbb{E} \rightarrow [\![\mathbb{B}]\!]$?

The Exceptional Implementation, Positive case

The really interesting case is the inductive part of CIC.

How to implement $[\mathbb{B}]_\varnothing : \mathbb{E} \to [\![\mathbb{B}]\!]$?

Could pose $[\![\mathbb{B}]\!] := \mathbb{B}$ and take an arbitrary boolean for $[\mathbb{B}]_\varnothing$...

... but that would not play well with computation, e.g. `catch`.

The Exceptional Implementation, Positive case

The really interesting case is the inductive part of CIC.

How to implement $[\mathbb{B}]_\varnothing : \mathbb{E} \to [\![\mathbb{B}]\!]$?

Could pose $[\![\mathbb{B}]\!] := \mathbb{B}$ and take an arbitrary boolean for $[\mathbb{B}]_\varnothing$...

... but that would not play well with computation, e.g. `catch`.

Worse, what about $[\bot]_\varnothing : \mathbb{E} \to [\![\bot]\!]$?

The Exceptional Implementation, Positive case

Very elegant solution: 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

Very elegant solution: 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.

$$[\![\Pi P : \mathbb{B} \to \square.\, P\ \texttt{true} \to P\ \texttt{false} \to \Pi b : \mathbb{B}.\, P\ b]\!]$$

$$\equiv \quad \Pi P : [\![\mathbb{B}]\!] \to [\![\square]\!].\, P\ [\texttt{true}] \to P\ [\texttt{false}] \to \Pi b : [\![\mathbb{B}]\!].\, P\ b$$

- 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 $[P\ b]_\varnothing\ e$

Shadok Logic Strikes Back

Theorem

The exceptional translation interprets all of CIC.

Shadok Logic Strikes Back

Theorem

The exceptional translation interprets all of CIC.

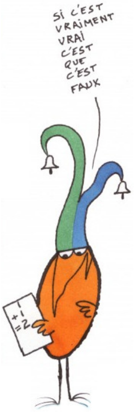
- ☺ A type theory with effects!
- ☺ Compiled away to CIC!
- ☺ Features full conversion
- ☺ Features full dependent elimination

Shadok Logic Strikes Back

Theorem

The exceptional translation interprets all of CIC.

- ☺ A type theory with effects!
- ☺ Compiled away to CIC!
- ☺ Features full conversion
- ☺ Features full dependent elimination



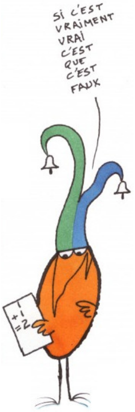
Shadok Logic Strikes Back

Theorem

The exceptional translation interprets all of CIC.

- ☺ A type theory with effects!
- ☺ Compiled away to CIC!
- ☺ Features full conversion
- ☺ Features full dependent elimination
- ☹ Ah, yeah, and also, the theory is inconsistent.

It suffices to raise an exception to inhabit any type.



Consistency: A Social Construct

An Impure Dependently-typed Programming Language

Do you whine about the fact that OCaml is logically inconsistent?

Consistency: A Social Construct

An Impure Dependently-typed Programming Language

Do you whine about the fact that OCaml is logically inconsistent?

Theorem (Exceptional Canonicity a.k.a. Progress a.k.a. Meaningless explanations)

If $\vdash_{\mathcal{T}_{\mathbb{E}}} M : \bot$, *then* $M \equiv \texttt{raise}\ \bot\ e$ *for some* $e : \mathbf{E}$.

Consistency: A Social Construct

An Impure Dependently-typed Programming Language

Do you whine about the fact that OCaml is logically inconsistent?

Theorem (Exceptional Canonicity a.k.a. Progress a.k.a. Meaningless explanations)

If $\vdash_{\mathcal{T}_{\mathbb{E}}} M : \bot$, *then* $M \equiv \mathtt{raise}\ \bot\ e$ *for some* $e : \mathbf{E}$.

A Safe Target Framework

You can still use the CIC target to prove properties about $\mathcal{T}_{\mathbb{E}}$ programs!

Consistency: A Social Construct

An Impure Dependently-typed Programming Language

Do you whine about the fact that OCaml is logically inconsistent?

Theorem (Exceptional Canonicity a.k.a. Progress a.k.a. Meaningless explanations)

If $\vdash_{\mathcal{T}_{\mathbb{E}}} M : \bot$, *then* $M \equiv \mathtt{raise}\ \bot\ e$ *for some* $e : \mathbf{E}$.

A Safe Target Framework

You can still use the CIC target to prove properties about $\mathcal{T}_{\mathbb{E}}$ programs!

Cliffhanger

You can prove that a program does not raise uncaught exceptions.

Consistency: A Social Construct

An Impure Dependently-typed Programming Language

Do you whine about the fact that OCaml is logically inconsistent?

Theorem (Exceptional Canonicity a.k.a. Progress a.k.a. Meaningless explanations)

If $\vdash_{\mathcal{T}_{\mathbb{E}}} M : \bot$, *then* $M \equiv \mathtt{raise}\ \bot\ e$ *for some* $e : \mathbf{E}$.

A Safe Target Framework

You can still use the CIC target to prove properties about $\mathcal{T}_{\mathbb{E}}$ programs!

Cliffhanger

You can prove that a program does not raise uncaught exceptions.

And now for a little ad before the second part of the show!

Informercial — Did You Know?

The exceptional translation is just a principled Friedman's A-translation!

Informercial — Did You Know?

The exceptional translation is just a principled Friedman's A-translation!

As such, it can be used for classical proof extraction.

Informative double-negation

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

Informercial — Did You Know?

The exceptional translation is just a principled Friedman's A-translation!

As such, it can be used for classical proof extraction.

Informative double-negation

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

First-order purification

If P is a Σ^0_1 type, then $\vdash_{\mathsf{CIC}} [\![P]\!] \leftrightarrow P + \mathbb{E}$.

Informercial — Did You Know?

The exceptional translation is just a principled Friedman's A-translation!

As such, it can be used for classical proof extraction.

Informative double-negation

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

First-order purification

If P is a Σ^0_1 type, then $\vdash_{\mathsf{CIC}} [\![P]\!] \leftrightarrow P + \mathbb{E}$.

Friedman's Trick in CIC

If P and Q are Σ^0_1 types, $\vdash_{\mathsf{CIC}} \Pi p : P.\, \neg\neg Q$ implies $\vdash_{\mathsf{CIC}} \Pi p : P.\, Q$.

Exception

Gotta catch 'em all!

If You Joined the Talk Recently

The exceptional type theory is logically inconsistent!

Cliffhanger (cont.)

You can prove that a program does not raise uncaught exceptions.

If You Joined the Talk Recently

The exceptional type theory is logically inconsistent!

Cliffhanger (cont.)

You can prove that a program does not raise uncaught exceptions.

Let's call **valid** a program in $\mathcal{T}_{\mathbb{E}}$ that "does not raise exceptions".

For instance,

- there is no valid proof of $\perp$
- the only valid booleans are `true` and `false`
- a function is valid if it produces a valid result out of a valid argument

If You Joined the Talk Recently

The exceptional type theory is logically inconsistent!

Cliffhanger (cont.)

You can prove that a program does not raise uncaught exceptions.

Let's call **valid** a program in $\mathcal{T}_{\mathbb{E}}$ that "does not raise exceptions".

For instance,

- there is no valid proof of $\bot$
- the only valid booleans are `true` and `false`
- a function is valid if it produces a valid result out of a valid argument

Validity is a type-directed notion!

The Curry-Howard-Shadok Correspondence

Let's locally write $M \Vdash A$ if M is valid at A.

The Curry-Howard-Shadok Correspondence

Let's locally write $M \Vdash A$ if M is valid at A.

$$f \Vdash A \rightarrow B \quad \equiv \quad \forall x : [\![A]\!]. \quad x \Vdash A \rightarrow f\,x \Vdash B$$

The Curry-Howard-Shadok Correspondence

Let's locally write $M \Vdash A$ if M is valid at A.

$$f \Vdash A \to B \quad \equiv \quad \forall x : [\![A]\!].\quad x \Vdash A \to f\,x \Vdash B$$

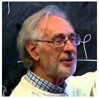
What? That's just **logical relations**.

The Curry-Howard-Shadok Correspondence

Let's locally write $M \Vdash A$ if M is valid at A.

$$f \Vdash A \rightarrow B \quad \equiv \quad \forall x : [\![A]\!]. \quad x \Vdash A \rightarrow f\,x \Vdash B$$

What? That's just **logical relations**.



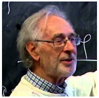
Come on. That's **intuitionistic realizability**.

The Curry-Howard-Shadok Correspondence

Let's locally write $M \Vdash A$ if M is valid at A.

$$f \Vdash A \to B \quad \equiv \quad \forall x : [\![A]\!]. \quad x \Vdash A \to f\,x \Vdash B$$

What? That's just **logical relations**.



Come on. That's **intuitionistic realizability**.

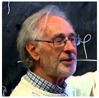
Fools ! That's **parametricity**.

The Curry-Howard-Shadok Correspondence

Let's locally write $M \Vdash A$ if M is valid at A.

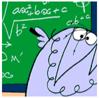
$$f \Vdash A \rightarrow B \quad \equiv \quad \forall x : [\![A]\!]. \quad x \Vdash A \rightarrow f\,x \Vdash B$$

What? That's just **logical relations**.



Come on. That's **intuitionistic realizability**.

Fools ! That's **parametricity**.



Zo!

Making Everybody Agree

It's actually folklore that these techniques are essentially the same.

Making Everybody Agree

It's actually folklore that these techniques are essentially the same.

And there is already a parametricity translation for CIC! (Bernardy-Lasson)

We just have to adapt it to our exceptional translation.

Making Everybody Agree

It's actually folklore that these techniques are essentially the same.

And there is already a parametricity translation for CIC! (Bernardy-Lasson)

We just have to adapt it to our exceptional translation.

Idea:

From $\vdash M : A$ produce **two** sequents
$$\begin{cases} \vdash_{\text{CIC}} [M] : [\![A]\!] \\ \qquad + \\ \vdash_{\text{CIC}} [M]_\varepsilon : [\![A]\!]_\varepsilon \ [M] \end{cases}$$

where $[\![A]\!]_\varepsilon : [\![A]\!] \to \square$ is the validity predicate.

Parametric Exceptional Translation (Sketch)

Most notably,

$$\begin{array}{lcl}
[\![\Pi x : A.\, B]\!]_\varepsilon\ f & \equiv & \Pi(x : [\![A]\!])\,(x_\varepsilon : [\![A]\!]_\varepsilon\ x).\, [\![B]\!]_\varepsilon\ (f\, x) \\
[\![\mathbb{B}]\!]_\varepsilon\ b & \cong & b = [\texttt{true}] + b = [\texttt{false}] \\
[\![\bot]\!]_\varepsilon\ s & \cong & \bot
\end{array}$$

Parametric Exceptional Translation (Sketch)

Most notably,

$$\begin{array}{lcl}
[\![\Pi x : A.\, B]\!]_\varepsilon\ f & \equiv & \Pi (x : [\![A]\!])\, (x_\varepsilon : [\![A]\!]_\varepsilon\ x).\, [\![B]\!]_\varepsilon\ (f\ x) \\
[\![\mathbb{B}]\!]_\varepsilon\ b & \cong & b = [\mathtt{true}] + b = [\mathtt{false}] \\
[\![\bot]\!]_\varepsilon\ s & \cong & \bot
\end{array}$$

Every pure term is now automatically parametric.

If $\Gamma \vdash_{\mathsf{CIC}} M : A$ then $[\![\Gamma]\!]_\varepsilon \vdash_{\mathsf{CIC}} [M]_\varepsilon : [\![A]\!]_\varepsilon\ [M]$.

A Few Nice Results

Let's call $\mathcal{T}^{p}_{\mathbb{E}}$ the resulting theory. It inherits a lot from CIC!

A Few Nice Results

Let's call $\mathcal{T}_{\mathbb{E}}^{p}$ the resulting theory. It inherits a lot from CIC!

Theorem (Consistency)

$\mathcal{T}_{\mathbb{E}}^{p}$ *is consistent.*

A Few Nice Results

Let's call $\mathcal{T}_{\mathbb{E}}^{p}$ the resulting theory. It inherits a lot from CIC!

Theorem (Consistency)

$\mathcal{T}_{\mathbb{E}}^{p}$ is consistent.

Theorem (Canonicity)

$\mathcal{T}_{\mathbb{E}}^{p}$ enjoys canonicity, i.e if $\vdash_{\mathcal{T}_{\mathbb{E}}^{p}} M : \mathbb{N}$ then $M \rightsquigarrow^{*} \bar{n} \in \bar{\mathbb{N}}$.

A Few Nice Results

Let's call $\mathcal{T}_{\mathbb{E}}^{p}$ the resulting theory. It inherits a lot from CIC!

Theorem (Consistency)

$\mathcal{T}_{\mathbb{E}}^{p}$ *is consistent.*

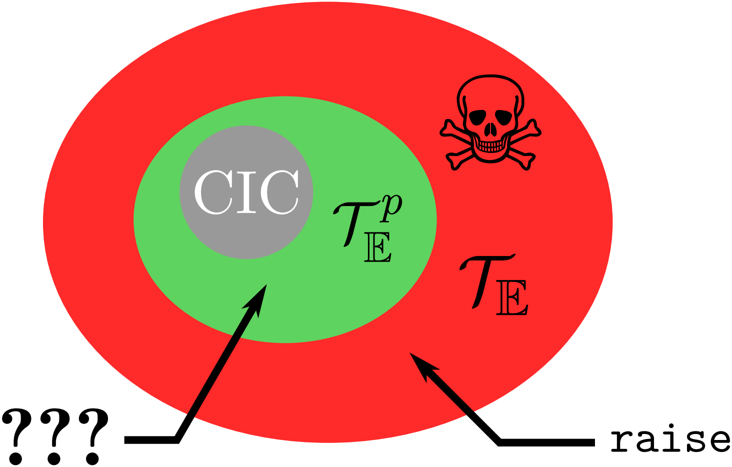
Theorem (Canonicity)

$\mathcal{T}_{\mathbb{E}}^{p}$ *enjoys canonicity, i.e if* $\vdash_{\mathcal{T}_{\mathbb{E}}^{p}} M : \mathbb{N}$ *then* $M \rightsquigarrow^{*} \bar{n} \in \bar{\mathbb{N}}$.

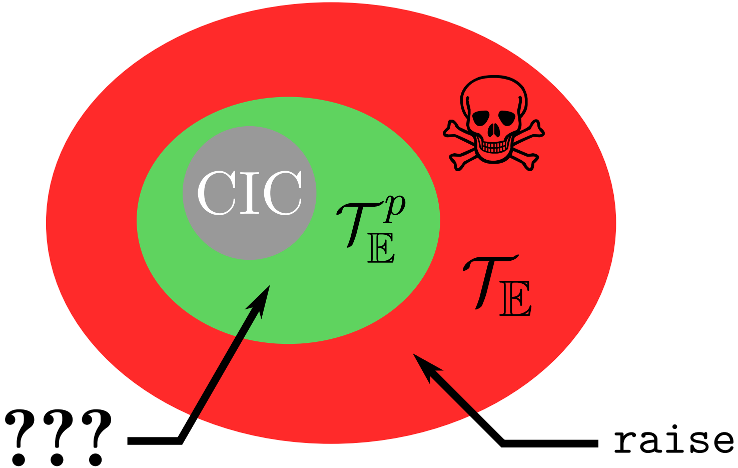
Theorem (Syntax)

$\mathcal{T}_{\mathbb{E}}^{p}$ *has decidable type-checking, strong normalization and whatnot.*

What If There Were No Cake?

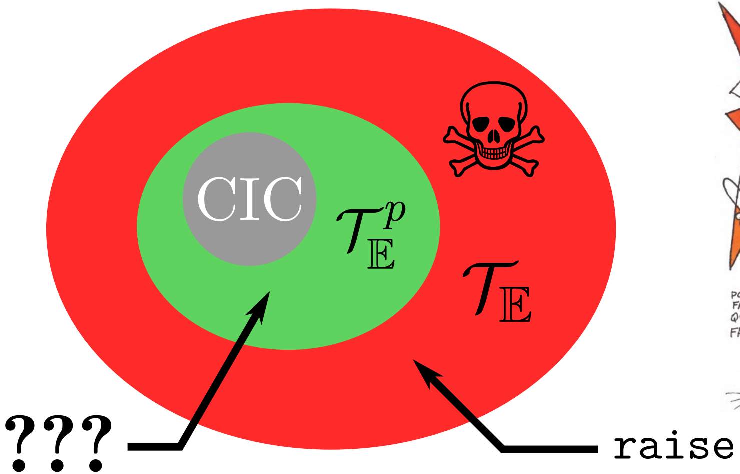


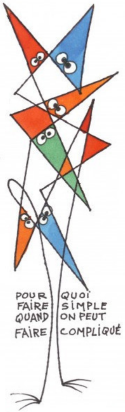
What If There Were No Cake?



Bernardy-Lasson parametricity is a conservative extension of CIC...

What If There Were No Cake?





Bernardy-Lasson parametricity is a conservative extension of CIC...

Less Is More

Spoiler

$\mathcal{T}^p_{\mathbb{E}}$ is **not** a conservative extension of CIC.

Less Is More

Spoiler

$\mathcal{T}_{\mathbb{E}}^{p}$ is **not** a conservative extension of CIC.

Intuitively,

- raising uncaught exceptions is forbidden in $\mathcal{T}_{\mathbb{E}}^{p}$

Less Is More

Spoiler

$\mathcal{T}_{\mathbb{E}}^{p}$ is **not** a conservative extension of CIC.

Intuitively,

- raising uncaught exceptions is forbidden in $\mathcal{T}_{\mathbb{E}}^{p}$
- … but you can still raise them locally
- … as long as you prove they don't escape!

Less Is More

Spoiler

$\mathcal{T}_{\mathbb{E}}^{p}$ is **not** a conservative extension of CIC.

Intuitively,

- raising uncaught exceptions is forbidden in $\mathcal{T}_{\mathbb{E}}^{p}$
- ... but you can still raise them locally
- ... as long as you prove they don't escape!

$\mathcal{T}_{\mathbb{E}}$ is the `unsafe` Coq fragment, and $\mathcal{T}_{\mathbb{E}}^{p}$ a semantical layer atop of it.

Less Is More

Spoiler

$\mathcal{T}_{\mathbb{E}}^{p}$ is **not** a conservative extension of CIC.

Intuitively,

- raising uncaught exceptions is forbidden in $\mathcal{T}_{\mathbb{E}}^{p}$
- … but you can still raise them locally
- … as long as you prove they don't escape!

$\mathcal{T}_{\mathbb{E}}$ is the `unsafe` Coq fragment, and $\mathcal{T}_{\mathbb{E}}^{p}$ a semantical layer atop of it.

Actually $\mathcal{T}_{\mathbb{E}}^{p}$ is the embodiement of Kreisel modified realizability in CIC.

Explaining the Analogy

	Kreisel realizability	$\mathcal{T}_{\mathbb{E}}^{p}$
Source theory	HA or HA^{ω}	CIC
Programming language	System T	$\mathcal{T}_{\mathbb{E}}$ (“unsafe Coq”)
Logical meta-theory	HA^{ω}	CIC

Explaining the Analogy

	Kreisel realizability	$\mathcal{T}_{\mathbb{E}}^{p}$
Source theory	HA or HA^ω	CIC
Programming language	System T	$\mathcal{T}_{\mathbb{E}}$ (“unsafe Coq”)
Logical meta-theory	HA^ω	CIC

Kreisel realizability extends arithmetic with essentially two principles.

- $\mathrm{AC}_{\mathbb{N}}$: $(\forall n : \mathbb{N}.\, \exists m : \mathbb{N}.\, P\ (m, n)) \rightarrow \exists f : \mathbb{N} \rightarrow \mathbb{N}.\, \forall n : \mathbb{N}.\, P\ (n, f\ n)$
- IP : $(\neg A \rightarrow \exists n : \mathbb{N}.\, P\ n) \rightarrow \exists n : \mathbb{N}.\, \neg A \rightarrow P\ n$

Choice

$$\mathsf{AC}_{\mathbb{N}} : (\forall n : \mathbb{N}.\, \exists m : \mathbb{N}.\, P\ (m, n)) \rightarrow \exists f : \mathbb{N} \rightarrow \mathbb{N}.\, \forall n : \mathbb{N}.\, P\ (n, f\ n)$$

Not much to say here.

In Kreisel realizability, $\mathsf{AC}_{\mathbb{N}}$ is a consequence of canonicity of System T.

Choice

$$\mathsf{AC}_{\mathbb{N}} : (\forall n : \mathbb{N}.\, \exists m : \mathbb{N}.\, P\ (m, n)) \to \exists f : \mathbb{N} \to \mathbb{N}.\, \forall n : \mathbb{N}.\, P\ (n, f\ n)$$

Not much to say here.

In Kreisel realizability, $\mathsf{AC}_{\mathbb{N}}$ is a consequence of canonicity of System T.

In $\mathcal{T}_{\mathbb{E}}^{p}$, $\mathsf{AC}_{\mathbb{N}}$ is a consequence of dependent elimination.

The latter is in turn meta-theoretically justified by canonicity.

Choice

$$\mathsf{AC}_{\mathbb{N}} : (\forall n : \mathbb{N}.\, \exists m : \mathbb{N}.\, P\ (m, n)) \rightarrow \exists f : \mathbb{N} \rightarrow \mathbb{N}.\, \forall n : \mathbb{N}.\, P\ (n, f\ n)$$

Not much to say here.

In Kreisel realizability, $\mathsf{AC}_{\mathbb{N}}$ is a consequence of canonicity of System T.

In $\mathcal{T}^{p}_{\mathbb{E}}$, $\mathsf{AC}_{\mathbb{N}}$ is a consequence of dependent elimination.

The latter is in turn meta-theoretically justified by canonicity.

In both cases, choice is built-in and a consequence of canonicity.

Independence of Premises

$$\mathsf{IP} : (\neg A \to \exists n : \mathbb{N}.\, P\ n) \to \exists n : \mathbb{N}.\, \neg A \to P\ n$$

That one is interesting! A unforeseen consequence of a subtle **bug**.

Kreisel's bug

Every type of realizers is inhabited. In particular, $[\![\bot]\!]_{\mathsf{KR}} \equiv \mathbb{N}$.

Independence of Premises

$$\mathsf{IP} : (\neg A \to \exists n : \mathbb{N}.\, P\ n) \to \exists n : \mathbb{N}.\, \neg A \to P\ n$$

That one is interesting! A unforeseen consequence of a subtle **bug**.

Kreisel's bug

Every type of realizers is inhabited. In particular, $[\![\bot]\!]_{\mathsf{KR}} \equiv \mathbb{N}$.

The realizer of IP critically relies on that!

Assuming System T had an empty type $\mathbb{0}$, and setting $[\![\bot]\!]_{\mathsf{KR}} \equiv \mathbb{0}$

- KR is still a model of HA
- KR still validates $\mathsf{AC}_{\mathbb{N}}$
- KR **doesn't** validate IP anymore

Volem Independència

$$\mathsf{IP} : (\neg A \rightarrow \exists n : \mathbb{N}.\, P\ n) \rightarrow \exists n : \mathbb{N}.\, \neg A \rightarrow P\ n$$

Theorem (CIC + IP)

$\mathcal{T}_{\mathbb{E}}^{p}$ *validates* IP, *owing to the fact that in* $\mathcal{T}_{\mathbb{E}}$, *every type is inhabited.*

Volem Independència

$$\mathsf{IP} : (\neg A \rightarrow \exists n : \mathbb{N}.\, P\ n) \rightarrow \exists n : \mathbb{N}.\, \neg A \rightarrow P\ n$$

Theorem (CIC + IP)

$\mathcal{T}^p_{\mathbb{E}}$ validates IP*, owing to the fact that in $\mathcal{T}_{\mathbb{E}}$, every type is inhabited.*

Proof (sketch).

In $\mathcal{T}_{\mathbb{E}}$, build a term `ip` : IP

- Given $f : \neg A \rightarrow \Sigma n : \mathbb{N}.\, P\ n$, apply it to `raise` $(\neg A)\ e$.
- If the returned integer is pure, return it with the associated proof.
- Otherwise, return a dummy integer and failing proof.

Easy to show that `ip` is actually valid in $\mathcal{T}^p_{\mathbb{E}}$. □

Another Result for Free

Recall Markov's principle:

$$\Pi P : \mathbb{N} \to \mathbb{B}.\, \neg\neg(\Sigma n : \mathbb{N}.\, P\ n = \mathtt{true}) \to \Sigma n : \mathbb{N}.\, P\ n = \mathtt{true} \qquad \text{(MP)}$$

Another Result for Free

Recall Markov's principle:

$$\Pi P : \mathbb{N} \to \mathbb{B}.\, \neg\neg(\Sigma n : \mathbb{N}.\, P\ n = \mathtt{true}) \to \Sigma n : \mathbb{N}.\, P\ n = \mathtt{true} \qquad \text{(MP)}$$

Kreisel's Razor

Pick two out of three: {canonicity, IP, MP}.

Another Result for Free

Recall Markov's principle:

$$\Pi P : \mathbb{N} \to \mathbb{B}.\, \neg\neg(\Sigma n : \mathbb{N}.\, P\ n = \mathtt{true}) \to \Sigma n : \mathbb{N}.\, P\ n = \mathtt{true} \qquad \text{(MP)}$$

Kreisel's Razor

Pick two out of three: {canonicity, IP, MP}.

$$\text{IP} + \text{MP} \Rightarrow \Pi P : \mathbb{N} \to \mathbb{B}.\, \Sigma n : \mathbb{N}.\, \Pi m : \mathbb{N}.\, P\ m = \mathtt{true} \to P\ n = \mathtt{true}$$

Together with canonicity, this solves the halting problem.

Another Result for Free

Recall Markov's principle:

$$\Pi P : \mathbb{N} \to \mathbb{B}.\, \neg\neg(\Sigma n : \mathbb{N}.\, P\ n = \texttt{true}) \to \Sigma n : \mathbb{N}.\, P\ n = \texttt{true} \qquad \text{(MP)}$$

Kreisel's Razor

Pick two out of three: {canonicity, IP, MP}.

IP + MP $\Rightarrow \Pi P : \mathbb{N} \to \mathbb{B}.\, \Sigma n : \mathbb{N}.\, \Pi m : \mathbb{N}.\, P\ m = \texttt{true} \to P\ n = \texttt{true}$

Together with canonicity, this solves the halting problem.

Corollary

$\not\vdash_{\mathcal{T}^p_{\mathbb{E}}}$ MP and thus $\not\vdash_{\mathsf{CIC}}$ MP.

(This was proved recently by Coquand-Mannaa, although in a completely different way.)

Function Intensionality

Another interesting consequence that is similar to what happens in KR.

- $\mathcal{T}_{\mathbb{E}}^{p}$ satisfies definitional η-expansion: $\lambda x : A.\, M\, x \equiv M$.
- But it violates function extensionality!

$$\vdash_{\mathcal{T}_{\mathbb{E}}^{p}} \Pi i : \mathbb{1}.\, i = \mathtt{tt} \qquad \text{and} \qquad \vdash_{\mathcal{T}_{\mathbb{E}}^{p}} (\lambda i : \mathbb{1}.\, i) \neq (\lambda i : \mathbb{1}.\, \mathtt{tt})$$

Function Intensionality

Another interesting consequence that is similar to what happens in KR.

- $\mathcal{T}_{\mathbb{E}}^{p}$ satisfies definitional η-expansion: $\lambda x : A.\, M\, x \equiv M$.
- But it violates function extensionality!

$$\vdash_{\mathcal{T}_{\mathbb{E}}^{p}} \Pi i : \mathbb{1}.\, i = \mathtt{tt} \qquad \text{and} \qquad \vdash_{\mathcal{T}_{\mathbb{E}}^{p}} (\lambda i : \mathbb{1}.\, i) \neq (\lambda i : \mathbb{1}.\, \mathtt{tt})$$

The reason is that there are invalid proofs of $\mathbb{1}$.

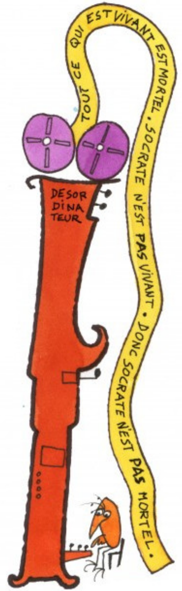
You cannot build them, but they exists as phantom arguments.

An Exceptional Coq Plugin

We implemented $\mathcal{T}_{\mathbb{E}}$ and $\mathcal{T}_{\mathbb{E}}^{p}$ in Coq in a plugin.

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

- Allows to add exceptions to Coq just today.
- Compile effectful terms on the fly.
- Allows to reason about them in Coq.
- Write mind-blowing low-level code!



If You Were Sleeping During The Talk

$\mathcal{T}_{\mathbb{E}}$, a type theory that allows failure!

- Inconsistent as a logical theory
- A dependently-typed effectful programming language
- Can still be used for proof extraction like Friedman's A-translation

If You Were Sleeping During The Talk

$\mathcal{T}_{\mathbb{E}}$, a type theory that allows failure!

- Inconsistent as a logical theory
- A dependently-typed effectful programming language
- Can still be used for proof extraction like Friedman's A-translation

$\mathcal{T}_{\mathbb{E}}^{p}$, a type theory that allows **local** failure!

- A safe layer atop $\mathcal{T}_{\mathbb{E}}$ that enforces consistency
- Strict superset of CIC: proves IP, $\neg$`funext`, disproves MP

If You Were Sleeping During The Talk

$\mathcal{T}_{\mathbb{E}}$, a type theory that allows failure!

- Inconsistent as a logical theory
- A dependently-typed effectful programming language
- Can still be used for proof extraction like Friedman's A-translation

$\mathcal{T}_{\mathbb{E}}^{p}$, a type theory that allows **local** failure!

- A safe layer atop $\mathcal{T}_{\mathbb{E}}$ that enforces consistency
- Strict superset of CIC: proves IP, $\neg$`funext`, disproves MP

Both of them justified by purely syntactical means!

If You Were Sleeping During The Talk

$\mathcal{T}_{\mathbb{E}}$, a type theory that allows failure!

- Inconsistent as a logical theory
- A dependently-typed effectful programming language
- Can still be used for proof extraction like Friedman's A-translation

$\mathcal{T}_{\mathbb{E}}^{p}$, a type theory that allows **local** failure!

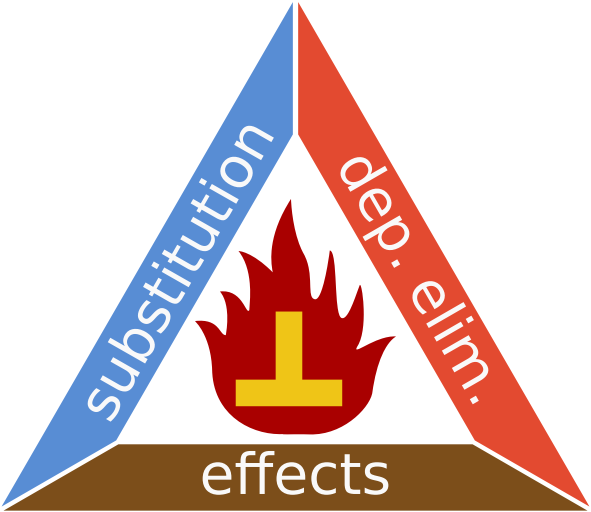
- A safe layer atop $\mathcal{T}_{\mathbb{E}}$ that enforces consistency
- Strict superset of CIC: proves IP, $\neg$`funext`, disproves MP

Both of them justified by purely syntactical means!

"The more it fails, the more likely it will eventually succeed."

Stepping Back

An Incompatibility



Conclusion

- You can add effects through syntactic models
- But you have to pick your side
- BTT is a CBN restriction that looks universal

Scribitur ad narrandum, non ad probandum.

Merci de votre attention.