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
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

\[ \vdash_{\text{CIC++}} M : A \quad \leadsto \quad \vdash_{\text{CIC}} [M] : [A] \]

« 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 ⊢ side-effects ⇔ new axioms
- Archetypical example: callcc and classical logic (Griffin, Krivine)
We already gave two instances of effectful variants of CIC.
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
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
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 $\leadsto$ use syntactic models (done)
- Stuff breaks $\leadsto$ 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 $\leadsto$ use syntactic models (done)
- Stuff breaks $\leadsto$ let’s concentrate on that today

We might lose part of our type-theoretic soul on the way.
Dependency entails one major difference with simpler types.
Dependency entails one major difference with simpler types.

Recall conversion:

\[
A \equiv_\beta B \quad \Gamma \vdash M : B \\
\Gamma \vdash M : A
\]
Dependency entails one major difference with simpler types.

Recall conversion:

\[ A \equiv_{\beta} B \quad \Gamma \vdash M : B \]
\[ \Gamma \vdash M : A \]

Bad news 1

Typing rules embed the dynamics of programs!
Conversion

Dependency entails one major difference with simpler types.

Recall conversion:

\[ A \equiv_\beta B \quad \Gamma \vdash M : B \]

\[ \Gamma \vdash M : A \]

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.
Effects make reduction strategies relevant.
Effects make reduction strategies relevant.

**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

\[
A \equiv_\beta B \quad \Gamma \vdash M : B
\]

\[
\Gamma \vdash M : A
\]
Problems

\[
A \equiv^\beta B \quad \Gamma \vdash M : B
\]

\[
\Gamma \vdash M : A
\]

Problem I

CIC has an CBN equational theory.

It’s unclear what you can do with CBV dependency...
Problems

\[
A \equiv^\beta B \\
\Gamma \vdash M : B \\
\hline
\Gamma \vdash M : A
\]

Problem I

CIC has an CBN equational theory.

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

\[
\text{bind}
\]

\[
T A \rightarrow (A \rightarrow T B) \rightarrow T B
\]

\[
\text{dbind}
\]

\[
\Pi(\hat{x} : T A). (\Pi(x : A). T (B x)) \rightarrow T (B ?)
\]
### Problem I

CIC has an CBN equational theory.

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

\[
\begin{align*}
\text{bind} & : T \, A \to (A \to T \, B) \to T \, B \\
\text{dbind} & : \Pi(\hat{x}: T \, A). (\Pi(x:A).T(B \, x)) \to T(B ?)
\end{align*}
\]

### Problem II

CBV monadic encodings don’t scale easily to dependent types.
Problems

\[ A \equiv_{\beta} B \quad \Gamma \vdash M : B \]
\[ \Gamma \vdash M : A \]

Problem I

CIC has an CBN equational theory.

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

\[ \text{bind} \quad T \ A \to (A \to T \ B) \to T \ B \]

\[ \Pi(\hat{x} : T \ A). (\Pi(x : A). T (B \ x)) \to T (B \ ?) \]

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
Reduction vs. Effects

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:

\[(\lambda x. M) \, N \equiv M\{x := N\} \leadsto \text{arbitrary substitution}\]

\[(\lambda b : \text{bool}. \, M) \, \text{fail} \leadsto \text{non-standard booleans}\]

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

\[(\lambda x. M) \, V \equiv M\{x := V\} \leadsto \text{substitute only values}\]

\[(\lambda b : \text{unit}. \, \text{fail} \, b) \leadsto \text{invalid } \eta\text{-rule}\]
Recall that dependent elimination is just the induction principle.

\[
\Gamma \vdash M : \mathbb{B} \quad \Gamma \vdash N_1 : P\{b := \text{true}\} \quad \Gamma \vdash N_2 : P\{b := \text{false}\}
\]

\[
\Gamma \vdash \text{if } M \text{ then } N_1 \text{ else } N_2 : P\{b := M\}
\]

This is a statement reflecting canonicity as an internal property in CIC.
Recall that dependent elimination is just the induction principle.

\[ \Gamma \vdash M : \mathbb{B} \quad \Gamma \vdash N_1 : P\{b := \text{true}\} \quad \Gamma \vdash N_2 : P\{b := \text{false}\} \]

\[ \Gamma \vdash \text{if } M \text{ then } N_1 \text{ else } N_2 : P\{b := M\} \]

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.**
Recall that dependent elimination is just the induction principle.

\[
\Gamma \vdash M : \mathbb{B} \quad \Gamma \vdash N_1 : P\{b := \text{true}\} \quad \Gamma \vdash N_2 : P\{b := \text{false}\}
\]

\[
\Gamma \vdash \text{if } M \text{ then } N_1 \text{ else } N_2 : P\{b := M\}
\]

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

But there are effectful closed booleans which are neither \text{true} nor \text{false}...

**Takeaway**

Dependent elimination is incompatible with CBN effects.

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 $\mathbb{R} : \Box$.

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

\[
\text{read} : \quad \mathbb{R} \\
\text{into} : \quad \Box \to \mathbb{R} \to \Box \\
\text{enter} : \quad \Pi(A : \Box). A \to \Pi r : \mathbb{R} \cdot \text{into } A \quad r
\]

(morally $\text{enter} : \quad \Pi(A : \Box). A \to \mathbb{R} \to A$)
The Reader Translation

Assume some fixed cell type $R : □$.

The reader translation extends CIC into CIC$_R$, with

\[
\begin{align*}
\text{read} & : \quad R \\
\text{into} & : \quad □ \to R \to □ \\
\text{enter} & : \quad Π(A : □). A \to Πr : R. \text{into } A \ r
\end{align*}
\]

(morally $\text{enter} : \quad Π(A : □). A \to R \to A$)

satisfying the expected definitional equations, e.g.

\[
\begin{align*}
\text{enter } R \ \text{read } M & \equiv M \\
\text{enter } R \ M \ \text{read} & \equiv M
\end{align*}
\]

Remember, we’re call-by-name...
Enters into

The \textit{into} is a mere typing artifact.

\[ \text{into}: \square \rightarrow \mathbb{R} \rightarrow \square \]

It has unfoldings on type formers:

\[ \text{into} (\Pi x: A. B) \; r \; \equiv \; \Pi x: A. \text{into} \; B \; r \]
\[ \text{into} \; \square \; r \; \equiv \; \square \]
\[ \ldots \]
Enters into

The into is a mere typing artifact.

\[
\text{into} : \Box \rightarrow \mathbb{R} \rightarrow \Box
\]

It has unfoldings on type formers:

\[
\text{into } (\Pi x : A. B) \ r \equiv \Pi x : A. \text{into } B \ r \\
\text{into } \Box \ r \equiv \Box \\
\]

... 

together with the following conversion:

\[
\text{into} \equiv \text{enter } \Box
\]

into is enter, but there is a typing loop.

Recall that:

\[
\text{enter} : \Pi (A : \Box). A \rightarrow \Pi r : \mathbb{R}. \text{into } A \ r
\]
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 $A : \Box$ into $[A]_r : \Box$
- Translate $M : A$ into $[M]_r : [A]_r$

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

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

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

- Translate \( A : \Box \) into \([A]_r : \Box\)
- Translate \( M : A \) into \([M]_r : [A]_r\)

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

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

Soundness

We have \( \Gamma \vdash M : A \) implies \( [\Gamma], r : \mathbb{R} \vdash [M]_r : [A]_r \).
PLT tells us we have to take $[A + B]_r \equiv [A] + [B]$.

$$
\begin{align*}
[A + B]_r & \equiv [A] + [B] \\
\text{[inl } M\text{]}_r & \equiv \text{inl } (\Pi s : \mathbb{R}. [M]_s) \\
\text{[inr } M\text{]}_r & \equiv \text{inr } (\Pi s : \mathbb{R}. [M]_s)
\end{align*}
$$
PLT tells us we have to take \( [A + B]_r \equiv [A] + [B] \).

\[
\begin{align*}
[A + B]_r & \equiv [A] + [B] \\
[inl \ M]_r & \equiv \text{inl} (\Pi s : \mathbb{R}. [M]_s) \\
[inr \ M]_r & \equiv \text{inr} (\Pi s : \mathbb{R}. [M]_s)
\end{align*}
\]

It’s possible to implement **non-dependent** pattern-matching as usual.
The Reader Implementation: Inductive Types

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

\[
\begin{align*}
[A + B]_r & \equiv [A] + [B] \\
[inl \ M]_r & \equiv \text{inl} \ (\Pi s : \mathbb{R}. [M]_s) \\
[inr \ M]_r & \equiv \text{inr} \ (\Pi s : \mathbb{R}. [M]_s)
\end{align*}
\]

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

\[
\begin{align*}
[\text{elim}_+]_r : [\Pi P : \Box. (A \to P) \to (B \to P) \to A + B \to P] & \equiv \Pi (P : \mathbb{R} \to \Box). \\
(\Pi s : \mathbb{R}. [A] \to P \ s) & \to (\Pi s : \mathbb{R}. [B] \to P \ s) \to (\mathbb{R} \to [A] + [B]) \to P \ r
\end{align*}
\]

\[
\begin{align*}
\text{elim}_+ \ P \ N_l \ N_r \ (\text{inl} \ M) & \equiv \ N_l \ M \\
\text{elim}_+ \ P \ N_l \ N_r \ (\text{inr} \ M) & \equiv \ N_r \ M
\end{align*}
\]
Unfortunately, It’s **not possible** to implement **dependent** elimination!

$$\left[\Pi P . (\Pi(x : A). P \text{ (inl } x)) \rightarrow (\Pi(y : B). P \text{ (inr } y)) \rightarrow \Pi b : A + B. P b\right]$$

$$\equiv$$

$$\Pi P : \mathbb{R} \rightarrow (\mathbb{R} \rightarrow [A] + [B]) \rightarrow \Box.$$  

$$\left((\Pi(s : \mathbb{R}) (x : [A]). P s (\lambda_\_ : \mathbb{R}. \text{inl } x)) \rightarrow (\Pi(s : \mathbb{R}) (y : [B]). P s (\lambda_\_ : \mathbb{R}. \text{inr } y)) \rightarrow \Pi(b : \mathbb{R} \rightarrow [A] + [B]). P r b\right)$$
Uh-oh

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

\[
[\Pi P. (\Pi(x : A). P (\text{inl } x)) \rightarrow (\Pi(y : B). P (\text{inr } y)) \rightarrow \Pi b : A + B. P b] \\
\equiv \\
\Pi P : \mathbb{R} \rightarrow (\mathbb{R} \rightarrow [A] + [B]) \rightarrow \square.
\]

\[
(\Pi(s : \mathbb{R}) (x : [A]). P s (\lambda_\_ : \mathbb{R}. \text{inl } x) ) \rightarrow \\
(\Pi(s : \mathbb{R}) (y : [B]). P s (\lambda_\_ : \mathbb{R}. \text{inr } y) ) \rightarrow \\
\Pi(b : \mathbb{R} \rightarrow [A] + [B]). P r b
\]

*P* only holds for two constant values but *b* can be anything!
Uh-oh

Unfortunately, It’s not possible to implement dependent elimination!

$$\left[ \Pi P. \left( \Pi (x : A). P \inl x \right) \rightarrow \left( \Pi (y : B). P \inr y \right) \right] \rightarrow \Pi b : A + B. P b$$

$$\equiv$$

$$\Pi P : \mathbb{R} \rightarrow (\mathbb{R} \rightarrow [A] + [B]) \rightarrow \Box.$$  

$$(\Pi (s : \mathbb{R}) \left( x : [A] \right). P \left( \lambda _{-} : \mathbb{R}. \inl x \right)) \rightarrow$$

$$(\Pi (s : \mathbb{R}) \left( y : [B] \right). P \left( \lambda _{-} : \mathbb{R}. \inr y \right)) \rightarrow$$

$$\Pi (b : \mathbb{R} \rightarrow [A] + [B]). P r b$$

$P$ 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 $\lceil \cdot \rceil_r$ predicates have the following type:

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

$$P : \mathbb{R} \rightarrow (\mathbb{R} \rightarrow [A] + [B]) \rightarrow \Box$$

Assume there is $\Phi : \mathbb{R} \rightarrow [A] + [B] \rightarrow \Box$ s.t.

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

In this case, induction principle becomes

$$(\Pi(s : \mathbb{R}) (x : [A]). \Phi \ s \ (\text{inl} \ x)) \rightarrow$$

$$(\Pi(s : \mathbb{R}) (y : [B]). \Phi \ s \ (\text{inr} \ y)) \rightarrow$$

$$\Pi(b : \mathbb{R} \rightarrow [A] + [B]). \Phi \ r \ (b \ r)$$

This is provable!
Not All Predicates are Equal

In general through $\lbrack \cdot \rbrack_r$ predicates have the following type:

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

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

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

In this case, induction principle becomes

$$(\Pi(s : \mathbb{R})(x : [A]). \Phi \ s \ (\text{inl} \ x)) \to$$

$$(\Pi(s : \mathbb{R})(y : [B]). \Phi \ s \ (\text{inr} \ y)) \to$$

$$\Pi(b : \mathbb{R} \to [A] + [B]). \Phi \ r \ (b \ r)$$

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:

$$P := \lambda b. \text{match } b \text{ with } \text{inl } x \rightarrow u_1 \mid \text{inr } y \rightarrow u_2$$

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

Pédrot (Gallinette)
Une Théorie des Types qui fait de l’effet
JFLA 2019
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.

$$P := \lambda b. \text{match } b \text{ with } \text{inl } x \rightarrow u_1 \mid \text{inr } y \rightarrow u_2$$

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

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?).

\[ \theta_{A+B} : A + B \to (A + B \to \Box) \to \Box \]
Storage Operators

Storage operator are a technique to implement CBV in CBN.

Originally from classical realizability, to implement \textit{induction} (ahem?).

\[ \theta_{A+B} : A + B \to (A + B \to \Box) \to \Box \]

- This is a CPS
Storage Operators

Storage operator are a technique to implement CBV in CBN.

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

\[ \theta_{A+B} : A + B \rightarrow (A + B \rightarrow \Box) \rightarrow \Box \]

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

\[ \theta_{A+B} b P := \text{match } b \text{ with } \text{inl } x \Rightarrow P(\text{inl } x) \mid \text{inr } y \Rightarrow P(\text{inr } y) \]
Storage Operators

Storage operators are a technique to implement CBV in CBN.

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

\[ \theta_{A+B} : A + B \rightarrow (A + B \rightarrow \Box) \rightarrow \Box \]

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

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

- In presence of dependent elimination,

\[ \vdash_{\text{CIC}} \theta_{A+B} \ b \ P = P \ b \]
Dependent elimination is **not valid** in general.

\[ \forall_{\text{CIC}_R} (\Pi(x : A). P(\text{inl } x)) \rightarrow (\Pi(y : B). P(\text{inr } y)) \rightarrow \Pi b : A + B. P b \]
Fixing Elimination

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

\[ \not \vdash_{\text{CIC}^R} (\Pi(x : A). P \text{ (inl } x)) \rightarrow (\Pi(y : B). P \text{ (inr } y)) \rightarrow \Pi b : A + B. P b \]

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

\[ \vdash_{\text{CIC}^R} (\Pi(x : A). P \text{ (inl } x)) \rightarrow (\Pi(y : B). P \text{ (inr } y)) \rightarrow \Pi b : A + B. \theta_{A+B} b P \]

Because \( \theta_{A+B} \) turns any predicate into an eager one.
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!
We restrict dependent elimination in the following way:

\[
\Gamma \vdash M : \mathbb{B} \quad \ldots \quad P \text{ linear in } b
\]

\[
\Gamma \vdash \text{if } M \text{ then } N_1 \text{ else } N_2 : P\{ b := M \}
\]

- 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:

\[ \Gamma \vdash M : B \quad \ldots \quad P \text{ linear in } b \]
\[ \Gamma \vdash \text{if } M \text{ then } N_1 \text{ else } N_2 : P\{b := M\} \]

- 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

\[ E : \Box \]
\[ \text{raise} : \Pi A : \Box. E \to A \]

As hinted before, we need to be call-by-name to feature full conversion.
The Exceptional Type Theory: Overview

The exceptional type theory extends vanilla CIC with

\[
\begin{align*}
\mathbf{E} &: \Box \\
\text{raise} &: \Pi A: \Box. \mathbf{E} \rightarrow A
\end{align*}
\]

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

\[
\begin{align*}
\text{raise } (\Pi x: A. B) e & \equiv \lambda x: A. \text{raise } B e \\
\text{match } (\text{raise } \mathcal{I} e) \text{ ret } P \text{ with } \vec{p} & \equiv \text{raise } (P \text{ (raise } \mathcal{I} e)) e
\end{align*}
\]

where \( P : \mathcal{I} \rightarrow \Box \).
The Exceptional Type Theory: Overview

The exceptional type theory extends vanilla CIC with

\[
\begin{align*}
E & : \Box \\
\text{raise} & : \Pi A : \Box. E \to A
\end{align*}
\]

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

\[
\begin{align*}
\text{raise} (\Pi x : A. B) e & \equiv \lambda x : A. \text{raise} B e \\
\text{match} (\text{raise} I e) \text{ ret } P \text{ with } \vec{p} & \equiv \text{raise} (P (\text{raise} I e)) e
\end{align*}
\]

where \( P : I \to \Box \).

Remark that in call-by-name, if \( M : A \to B \), in general

\[
M (\text{raise} A e) \neq \text{raise} B e
\]

for otherwise we would not have \((\lambda x : A. M) N \equiv M\{x := N\}\).
Catch Me If You Can

Remember that on functions:

\[ \text{raise } (\Pi x : A. B) \ e \ \equiv \ \lambda x : A. \text{raise } B \ e \]

It means catching exceptions is limited to positive datatypes!
Catch Me If You Can

Remember that on functions:

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

It means catching exceptions is limited to positive datatypes!

For inductive types, this is a generalized induction principle.

\[
\begin{align*}
\text{catch}_B & : \Pi P : B \to \square. \\
& \quad P \ \text{true} \to \\
& \quad P \ \text{false} \to \\
& \quad (\Pi e : E. P (\text{raise } B \ e)) \to \\
& \quad \Pi b : B. P \ b
\end{align*}
\]

where

\[
\begin{align*}
\text{catch}_B \ P \ p_t \ p_f \ p_e \ \text{true} & \equiv \ p_t \\
\text{catch}_B \ P \ p_t \ p_f \ p_e \ \text{false} & \equiv \ p_f \\
\text{catch}_B \ P \ p_t \ p_f \ p_e \ (\text{raise } B \ e) & \equiv \ p_e \ e
\end{align*}
\]
Let’s implement the exceptional type theory into CIC!
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 $\mathcal{T}_E$ to disambiguate it from CIC.
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 $T_E$ to disambiguate it from CIC.

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

$$\vdash_{\text{CIC}} E : \Box$$
The Exceptional Implementation, Negative case

Intuition: $\vdash_{\Gamma_E} A : \Box \implies \vdash_{\text{CIC}} [A] : \Sigma A : \Box. E \to A.$

Every exceptional type comes with its own implementation of failure!
The Exceptional Implementation, Negative case

Intuition: \( \Gamma \vdash T \quad \leadsto \quad \Gamma \vdash_{\text{CIC}} [A] : \Sigma A : \Box. \mathbb{E} \to A. \)

Every exceptional type comes with its own implementation of failure!

\[
[A] : \Box := \pi_1 [A] \quad \text{and} \quad [A]_\emptyset : \mathbb{E} \to [A] := \pi_2 [A]
\]

\[
\begin{align*}
[\Box] & \equiv \Sigma A : \Box. \mathbb{E} \to A \\
[\Box]_\emptyset e & \equiv \ldots \\
[\Pi x : A. \ B] & \equiv \Pi x : [A]. [B] \\
[\Pi x : A. \ B]_\emptyset e & \equiv \lambda x : [A]. [B]_\emptyset e
\end{align*}
\]
The Exceptional Implementation, Negative case

Intuition: \( \vdash_{\mathcal{E}} A : \Box \implies \vdash_{\text{ClC}} [A] : \Sigma A : \Box. \mathbb{E} \rightarrow A. \)

Every exceptional type comes with its own implementation of failure!

\[ [A] : \Box := \pi_1 [A] \quad \text{and} \quad [A]_{\Box} : \mathbb{E} \rightarrow [A] := \pi_2 [A] \]

\[
\begin{align*}
[\Box] & \equiv \Sigma A : \Box. \mathbb{E} \rightarrow A \\
[\Box]_{\Box} e & \equiv \ldots \\
[\Pi x : A. B] & \equiv \Pi x : [A]. [B] \\
[\Pi x : A. B]_{\Box} e & \equiv \lambda x : [A]. [B]_{\Box} e \\
[x] & \equiv x \\
[M N] & \equiv [M] [N] \\
[\lambda x : A. M] & \equiv \lambda x : [A]. [M]
\end{align*}
\]
The Exceptional Implementation, Negative case

Intuition: $\vdash_{T_E} A : \Box \leadsto \vdash_{\text{CIC}} [A] : \Sigma A : \Box. \mathbb{E} \rightarrow A.$

Every exceptional type comes with its own implementation of failure!

$$[A] : \Box := \pi_1 [A] \quad \text{and} \quad [A]_{\emptyset} : \mathbb{E} \rightarrow [A] := \pi_2 [A]$$

$$\begin{align*}
\Box & \equiv \Sigma A : \Box. \mathbb{E} \rightarrow A \\
\Box_{\emptyset} e & \equiv \ldots \\
[\Pi x : A. B] & \equiv \Pi x : [A]. [B] \\
[\Pi x : A. B]_{\emptyset} e & \equiv \lambda x : [A]. [B]_{\emptyset} e \\
x & \equiv x \\
[M \ N] & \equiv [M] \ [N] \\
[\lambda x : A. M] & \equiv \lambda x : [A]. [M]
\end{align*}$$

If $\Gamma \vdash_{\text{CIC}} M : A$ then $[\Gamma] \vdash_{\text{CIC}} [M] : [A].$
The Exceptional Implementation, Failure

It is straightforward to implement the failure operation.

\[
E : \Box \\
\text{raise} : \Pi A : \Box. E \rightarrow A
\]
The Exceptional Implementation, Failure

It is straightforward to implement the failure operation.

\[
\begin{align*}
E & : \Box \\
\text{raise} & : \Pi A : \Box. E \to A
\end{align*}
\]

\[
\begin{align*}
[E] & : \Sigma A : \Box. E \to A \\
[E] & := (E, \lambda e : E. e) \\
\text{[raise]} & : \Pi A_0 : (\Sigma A : \Box. E \to A). E \to \pi_1 A_0 \\
\text{[raise]} & := \pi_2
\end{align*}
\]
The Exceptional Implementation, Failure

It is straightforward to implement the failure operation.

\[
\begin{align*}
\mathbf{E} & : \Box \\
\text{raise} & : \Pi A : \Box. \mathbf{E} \rightarrow A
\end{align*}
\]

\[
\begin{align*}
[\mathbf{E}] & : \Sigma A : \Box. \mathbf{E} \rightarrow A \\
[\mathbf{E}] & := (\mathbf{E}, \lambda e : \mathbf{E}. e)
\end{align*}
\]

\[
\begin{align*}
[\text{raise}] & : \Pi A_0 : (\Sigma A : \Box. \mathbf{E} \rightarrow A). \mathbf{E} \rightarrow \pi_1 A_0 \\
[\text{raise}] & := \pi_2
\end{align*}
\]

Computational rules trivially hold!

\[
\begin{align*}
[\text{raise} \ (\Pi x : A. B) \ e] \equiv & \ [\lambda x : A. \text{raise} \ B \ e] \\
\equiv & \ \pi_2 ((\Pi x : [A]. [B]), (\lambda (e : \mathbf{E}) (x : [A]). \pi_2 [B] e)) [e] \\
\equiv & \ \lambda x : [A]. \pi_2 [B] [e]
\end{align*}
\]
The really interesting case is the inductive part of CIC.

How to implement $[B]_{\emptyset} : E \rightarrow [B]$?
The really interesting case is the inductive part of CIC.

How to implement $[B]_{\emptyset} : E \rightarrow [B]$?

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

... but that would not play well with computation, e.g. `catch`.
The really interesting case is the inductive part of CIC.

How to implement \([B]_\emptyset : \text{E} \rightarrow [B]?)

Could pose \([B] := B\) and take an arbitrary boolean for \([B]_\emptyset\)...

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

Worse, what about \([\bot]_\emptyset : \text{E} \rightarrow [\bot]?)
Very elegant solution: add a default case to every inductive type!

\[
\text{Inductive } [\mathcal{B}] := [\text{true}]: [\mathcal{B}] \mid [\text{false}]: [\mathcal{B}] \mid \mathcal{B}_\emptyset : \mathcal{E} \rightarrow [\mathcal{B}]
\]
Very elegant solution: add a default case to every inductive type!

\[
\text{Inductive } [B] := [\text{true}]:[B] \mid [\text{false}]:[B] \mid B\emptyset : E \to [B]
\]

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

\[
[\Pi P : B \to \Box. P \text{ true} \to P \text{ false} \to \Pi b : B. P \ b]
\]
\[
\equiv \Pi P : [B] \to [\Box]. P [\text{true}] \to P [\text{false}] \to \Pi b : [B]. P \ b
\]

- If \( b \) is \([\text{true}]\), use first hypothesis
- If \( b \) is \([\text{false}]\), use second hypothesis
- If \( b \) is an error \( B\emptyset \ e \), \textbf{reraise} \( e \) using \([P \ b] \emptyset \ e\)
Theorem

*The exceptional translation interprets all of CIC.*
Theorem

The exceptional translation interprets all of CIC.

😊 A type theory with effects!
😊 Compiled away to CIC!
😊 Features full conversion
😊 Features full dependent elimination
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.
Theorem (Exceptional Canonicity a.k.a. Progress a.k.a. Meaningless explanations)
If \( \vdash T \varepsilon M : \bot \), then \( M \equiv \text{raise} \bot e \) for some \( e : E \).

A Safe Target Framework
You can still use the CIC target to prove properties about TE programs!
An Impure Dependently-typed Programming Language

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

\[ \text{Theorem (Exceptional Canonicity a.k.a. Progress a.k.a. Meaningless explanations)} \]

If \( \vdash \tau_M : \bot \), then \( M \equiv \text{raise } \bot \, e \) for some \( e : E \).
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)

\[ \text{If } \vdash_{\mathcal{T}_E} M : \bot, \text{ then } M \equiv \text{raise } \bot \text{ for some } e : \mathcal{E}. \]

A Safe Target Framework

You can still use the CIC target to prove properties about \( \mathcal{T}_E \) 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 $\vdash_{\mathcal{E}} M : \bot$, then $M \equiv \text{raise } \bot e$ for some $e : \mathcal{E}$.

A Safe Target Framework

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

Cliffhanger

You can prove that a program does not raise uncaught exceptions.
Consistency: A Social Construct

An Impure Dependent-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 $\vdash_{\mathcal{T}_E} M : \bot$, then $M \equiv \text{raise } \bot \; e$ for some $e : \mathcal{E}$.

A Safe Target Framework

You can still use the CIC target to prove properties about $\mathcal{T}_E$ 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

$$[
eg

\neg A] \cong ([A] \rightarrow \mathbb{E}) \rightarrow \mathbb{E}$$
The exceptional translation is just a principled Friedman’s $A$-translation!

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

Informative double-negation

$$\neg\neg A \cong ([A] \rightarrow \bot) \rightarrow \bot$$

First-order purification

If $P$ is a $\Sigma^0_1$ type, then $\vdash_{\text{CIC}} [P] \leftrightarrow P + \bot$. 
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

\[[\neg\neg A] \cong ([A] \to \bot) \to \bot\]

First-order purification

If $P$ is a $\Sigma^0_1$ type, then $\vdash_{\text{CIC}} [P] \iff P + \bot$.

Friedman’s Trick in CIC

If $P$ and $Q$ are $\Sigma^0_1$ types, $\vdash_{\text{CIC}} \Pi p : P. \neg\neg Q$ implies $\vdash_{\text{CIC}} \Pi p : P. Q$. 
Exception
Gotta catch 'em all!
The exceptional type theory is logically inconsistent!

Cliffhanger (cont.)

You can prove that a program does not raise uncaught exceptions.
The exceptional type theory is logically inconsistent!

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

Let’s call valid a program in $\mathcal{T}_E$ 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
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 $\mathcal{T}_E$ 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!
Let’s locally write $\vdash M A$ if $M$ is valid at $A$. 
The Curry-Howard-Shadok Correspondence

Let’s locally write $M \vdash A$ if $M$ is valid at $A$.

\[
f \vdash A \rightarrow B \equiv \forall x : [A]. \ x \vdash A \rightarrow f \ x \vdash B\]
The Curry-Howard-Shadok Correspondence

Let’s locally write $M \vdash A$ if $M$ is valid at $A$.

$$f \vdash A \rightarrow B \equiv \forall x : [A]. x \vdash A \rightarrow f\,x \vdash B$$

What? That’s just logical relations.
Let’s locally write $M \vdash A$ if $M$ is valid at $A$.

\[
f \vdash A \to B \equiv \forall x : [A]. \; x \vdash A \to f x \vdash B
\]

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 $A$.

\[
f \vdash A \rightarrow B \equiv \forall x : \llbracket A \rrbracket. \quad x \vdash A \rightarrow f x \vdash B
\]

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

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

Fools! That’s **parametricity**.
The Curry-Howard-Shadok Correspondence

Let’s locally write $M \vdash A$ if $M$ is valid at $A$.

$$f \vdash A \rightarrow B \equiv \forall x : [A]. \ x \vdash A \rightarrow f \ x \vdash B$$

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

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

Fools! That’s **parametricity**.

Zo!
It’s actually folklore that these techniques are essentially the same.

Idea:

\[
\begin{align*}
\Gamma \vdash M : A \\
\Gamma \vdash \text{CIC}[M] : [A]_+ \\
\Gamma \vdash \text{CIC}[M] : [A]_\varepsilon \varepsilon [M]
\end{align*}
\]

where \([A]_\varepsilon : [A]_\rightarrow \square\) is the validity predicate.
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.
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

\[
\begin{align*}
\vdash_{CIC} [M] : [A] \\
+ \\
\vdash_{CIC} [M]_\varepsilon : [A]_\varepsilon [M]
\end{align*}
\]

where \([A]_\varepsilon : [A] \rightarrow \square\) is the validity predicate.
Most notably,

\[
[\Pi x : A. B]_\varepsilon f \equiv \Pi(x : [A]) (x_\varepsilon : [A]_\varepsilon x). [B]_\varepsilon (f x)
\]

\[
[B]_\varepsilon b \equiv b = [\text{true}] + b = [\text{false}]
\]

\[
[\bot]_\varepsilon s \equiv \bot
\]
Most notably,

\[
[\Pi x : A. B]_\varepsilon f \equiv \Pi(x : [A])(x_\varepsilon : [A]_\varepsilon x). [B]_\varepsilon (f x)
\]

\[
[B]_\varepsilon b \equiv b = [\text{true}] + b = [\text{false}]
\]

\[
[\bot]_\varepsilon s \equiv \bot
\]

Every pure term is now automatically parametric.

If \( \Gamma \vdash_{\text{CIC}} M : A \) then \( [\Gamma]_\varepsilon \vdash_{\text{CIC}} [M]_\varepsilon : [A]_\varepsilon [M] \).
Let’s call $\mathcal{T}_E^p$ the resulting theory. It inherits a lot from CIC!
Let’s call $\mathcal{T}^p_E$ the resulting theory. It inherits a lot from CIC!

**Theorem (Consistency)**

$\mathcal{T}^p_E$ is consistent.
A Few Nice Results

Let’s call $T^p_E$ the resulting theory. It inherits a lot from CIC!

Theorem (Consistency)

$T^p_E$ is consistent.

Theorem (Canonicity)

$T^p_E$ enjoys canonicity, i.e if $\vdash T^p_E M : \mathbb{N}$ then $M \rightsquigarrow^* \bar{n} \in \bar{\mathbb{N}}$. 
A Few Nice Results

Let’s call $\mathcal{T}_E^p$ the resulting theory. It inherits a lot from CIC!

**Theorem (Consistency)**

$\mathcal{T}_E^p$ is consistent.

**Theorem (Canonicity)**

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

**Theorem (Syntax)**

$\mathcal{T}_E^p$ has decidable type-checking, strong normalization and whatnot.
What If There Were No Cake?

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

Pédrot (Gallinette)
Une Théorie des Types qui fait de l’effet
JFLA 2019

??? raise
Bernardy-Lasson parametricity is a conservative extension of CIC...
Bernardy-Lasson parametricity is a conservative extension of CIC...
Less Is More

\[ \mathcal{T}^p_E \text{ is not a conservative extension of CIC.} \]
$\mathcal{T}_E^p$ is not a conservative extension of CIC.

Intuitively,

- raising uncaught exceptions is forbidden in $\mathcal{T}_E^p$
Less Is More

$\mathcal{T}_p$ is not a conservative extension of CIC.

Intuitively,

- raising uncaught exceptions is forbidden in $\mathcal{T}_p$
- ... but you can still raise them locally
- ... as long as you prove they don’t escape!
Less Is More

### Spoiler

\( \mathcal{T}_E^p \) is not a conservative extension of CIC.

Intuitively,

- raising uncaught exceptions is forbidden in \( \mathcal{T}_E^p \)
- ... but you can still raise them locally
- ... as long as you prove they don’t escape!

\( \mathcal{T}_E \) is the unsafe Coq fragment, and \( \mathcal{T}_E^p \) a semantical layer atop of it.
\( \mathcal{T}_E^p \) is not a conservative extension of CIC.

Intuitively,

- raising uncaught exceptions is forbidden in \( \mathcal{T}_E^p \)
- ... but you can still raise them locally
- ... as long as you prove they don’t escape!

\( \mathcal{T}_E \) is the unsafe Coq fragment, and \( \mathcal{T}_E^p \) a semantical layer atop of it.

Actually \( \mathcal{T}_E^p \) is the embodiment of Kreisel modified realizability in CIC.
### Explaining the Analogy

<table>
<thead>
<tr>
<th>Source theory</th>
<th>Kreisel realizability</th>
<th>$\mathcal{T}_E^p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>HA or HA$^\omega$</td>
<td>CIC</td>
<td>$\mathcal{T}_E$ (&quot;unsafe Coq&quot;)</td>
</tr>
<tr>
<td>Programming language</td>
<td>System T</td>
<td></td>
</tr>
<tr>
<td>Logical meta-theory</td>
<td>HA$^\omega$</td>
<td>CIC</td>
</tr>
</tbody>
</table>

Kreisel realizability extends arithmetic with essentially two principles:

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

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

Pédrot (Gallinette)
Explaining the Analogy

<table>
<thead>
<tr>
<th>Source theory</th>
<th>Kreisel realizability</th>
<th>Source theory</th>
<th>Logical meta-theory</th>
</tr>
</thead>
<tbody>
<tr>
<td>HA or HA$^\omega$</td>
<td>HA$^\omega$</td>
<td>HA or HA$^\omega$</td>
<td>HA$^\omega$</td>
</tr>
<tr>
<td>System T</td>
<td>CIC</td>
<td>$\mathcal{T}_E$ (“unsafe Coq”)</td>
<td>CIC</td>
</tr>
</tbody>
</table>

Kreisel realizability extends arithmetic with essentially two principles.

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

\[ AC_N : (\forall n : \mathbb{N}. \exists m : \mathbb{N}. P (m, n)) \rightarrow \exists f : \mathbb{N} \rightarrow \mathbb{N}. \forall n : \mathbb{N}. P (n, f n) \]

Not much to say here.

In Kreisel realizability, \( AC_N \) is a consequence of canonicity of System T.
Choice

\[ AC_\mathbb{N} : (\forall n : \mathbb{N} \exists m : \mathbb{N} \ P (m, n)) \rightarrow \exists f : \mathbb{N} \rightarrow \mathbb{N} . \forall n : \mathbb{N} . P (n, f n) \]

Not much to say here.

In Kreisel realizability, \( AC_\mathbb{N} \) is a consequence of canonicity of System T.

In \( T^p_\mathcal{E} \), \( AC_\mathbb{N} \) is a consequence of dependent elimination.

The latter is in turn meta-theoretically justified by canonicity.
AC$_N$ : $(\forall n : \mathbb{N}. \exists m : \mathbb{N}. P (m, n)) \rightarrow \exists f : \mathbb{N} \rightarrow \mathbb{N}. \forall n : \mathbb{N}. P (n, f n)$

Not much to say here.

In Kreisel realizability, AC$_N$ is a consequence of canonicity of System T.

In $\mathcal{T}_E^p$, AC$_N$ 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

\[ IP : (\neg A \rightarrow \exists n : \mathbb{N}. P \ n) \rightarrow \exists n : \mathbb{N}. \neg A \rightarrow P \ n \]

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

Kreisel’s bug

Every type of realizers is inhabited. In particular, \( [\bot]_{KR} \equiv \mathbb{N} \).
Independence of Premises

\[ \text{IP : } (\neg A \rightarrow \exists n : \mathbb{N}. P \ n) \rightarrow \exists n : \mathbb{N}. \neg A \rightarrow P \ n \]

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

Kreisel’s bug

Every type of realizers is inhabited. In particular, \( \llbracket \bot \rrbracket_{KR} \equiv \mathbb{N} \).

The realizer of IP critically relies on that!

Assuming System T had an empty type \( \emptyset \), and setting \( \llbracket \bot \rrbracket_{KR} \equiv \emptyset \)

- KR is still a model of HA
- KR still validates \( AC_{\mathbb{N}} \)
- KR doesn’t validate IP anymore
Theorem (CIC + IP)

$\mathcal{T}_E^p$ validates IP, owing to the fact that in $\mathcal{T}_E$, every type is inhabited.
Theorem (CIC + IP)

\( \mathcal{T}_E^p \) validates IP, owing to the fact that in \( \mathcal{T}_E \), every type is inhabited.

Proof (sketch).

In \( \mathcal{T}_E \), build a term \( \text{ip} : \text{IP} \)

- Given \( f : \neg A \to \Sigma n : \mathbb{N}. P n \), 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 \( \text{ip} \) is actually valid in \( \mathcal{T}_E^p \).
Another Result for Free

Recall Markov’s principle:

\[ \Pi P : \mathbb{N} \rightarrow \mathbb{B}. \neg \neg (\Sigma n : \mathbb{N}. P n = \text{true}) \rightarrow \Sigma n : \mathbb{N}. P n = \text{true} \quad (\text{MP}) \]
Recall Markov’s principle:

\[ \Pi P : \mathbb{N} \to \mathbb{B}. \neg \neg (\Sigma n : \mathbb{N}. P n = \text{true}) \to \Sigma n : \mathbb{N}. P n = \text{true} \quad \text{(MP)} \]

**Kreisel’s Razor**

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

Another Result for Free

Recall Markov’s principle:

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

Kreisel’s Razor

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

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

Together with canonicity, this solves the halting problem.
Another Result for Free

Recall Markov’s principle:

\[ \forall P : \mathbb{N} \rightarrow \mathbb{B}. \neg \neg (\sum n : \mathbb{N}. P \ n = \text{true}) \rightarrow \sum n : \mathbb{N}. P \ n = \text{true} \]  

(MP)

Kreisel’s Razor

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

\[ \text{IP} + \text{MP} \Rightarrow \forall P : \mathbb{N} \rightarrow \mathbb{B}. \sum n : \mathbb{N}. \forall m : \mathbb{N}. P \ m = \text{true} \rightarrow P \ n = \text{true} \]

Together with canonicity, this solves the halting problem.

Corollary

\[ \vdash_{\text{T}^p} \text{MP} \text{ and thus } \vdash_{\text{CIC}} \text{MP}. \]

(This was proved recently by Coquand-Mannaa, although in a completely different way.)
Another interesting consequence that is similar to what happens in KR.

- $T^p_E$ satisfies definitional $\eta$-expansion: $\lambda x : A. M x \equiv M$.
- But it violates function extensionality!

$$\vdash_{T^p_E} \Pi i : 1. i = \text{tt} \quad \text{and} \quad \vdash_{T^p_E} (\lambda i : 1. i) \neq (\lambda i : 1. \text{tt})$$
Function Intensionality

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

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

$\vdash \mathcal{T}_{IE}^p \Pi i : 1. i = \mathsf{tt}$ and $\vdash \mathcal{T}_{IE}^p (\lambda i : 1. i) \neq (\lambda i : 1. \mathsf{tt})$

The reason is that there are invalid proofs of $1$.

You cannot build them, but they exists as phantom arguments.
An Exceptional Coq Plugin

We implemented $\mathcal{T}_E$ and $\mathcal{T}_E^p$ 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

$\mathcal{T}_E$, 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

\( \mathcal{T}_E \), 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 \( \Lambda \)-translation

\( \mathcal{T}_E^p \), a type theory that allows \textbf{local} failure!

- A safe layer atop \( \mathcal{T}_E \) that enforces consistency
- Strict superset of CIC: proves IP, \( \neg \text{funext} \), disproves MP
If You Were Sleeping During The Talk

$\mathcal{T}_E$, 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

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

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

Both of them justified by purely syntactical means!
If You Were Sleeping During The Talk

$\mathcal{T}_E$, 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 $\lambda$-translation

$\mathcal{T}^p_E$, a type theory that allows local failure!

- A safe layer atop $\mathcal{T}_E$ 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

substitution

dep. elim.

effects

Pédrot (Gallinette)  Une Théorie des Types qui fait de l’effet  JFLA 2019  55 / 56
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.