Failure is Not an Option
The Curry-Howard-Shadok correspondence

Pierre-Marie Pédrot
joint work with Nicolas Tabareau

Max Planck Institute for Software Systems

Séminaire PPS
Les Fondements de la Pensée SHADOK

JE NE PENSE PAS DONC JE SUIS, ET RÉCIPROQUEMENT
S'IL N'YA PAS DE SOLUTION C'EST QU'IL N'YA PAS DE PROBLÈME
POUR FAIRE QUAND ON PEUT COMPLIQUÉ
CE N'EST QU'EN ÉSSAYANT CONTINUÉMENT QUE L'ON FINIT PAR RÉUSSIR
... PLUS GARANTIE PLUS ON A DE CHANCES QUE ÇA MARCHÉ
SI C'EST VRAIMENT VRAI C'EST QUE ÇA EST FAUX
QUI EST VIVANT SOCIÉTÉ N'EST PAS VIVANT, DONC SOCRATE N'EST PAS MORTEL

Jacques Rouxel
It’s time to CIC ass and chew bubble-gum

CIC, the Calculus of Inductive Constructions.
It’s time to CIC ass and chew bubble-gum

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
It’s time to CIC ass and chew bubble-gum

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
It’s time to CIC ass and chew bubble-gum

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
Un Coq qui fait de l'effet

My research has been focussed on the extension of CIC with side-effects.
My research has been focussed on the extension of CIC with **side-effects**.

**To Program More!**

- Obviously you want effects to program
- E.g. state, exceptions, non-termination, continuations...
My research has been focused on the extension of CIC with **side-effects**.

**To Program More!**

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

**To Prove More!**

- A well-known fact here at PPS
- Curry-Howard ⊢ side-effects ⇔ new axioms
- Archetypical example: callcc and classical logic (Griffin, Krivine)
Summary of the Previous Episodes

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
Summary of the Previous Episodes

We already gave two instances of effectual 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
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
Recall that dependent elimination for booleans amounts to

\[
\begin{align*}
\Gamma \vdash M : \mathbb{B} & \quad \Gamma \vdash N_1 : P\{\text{true}\} & \quad \Gamma \vdash N_2 : P\{\text{false}\} \\
\hline \\
\Gamma \vdash \text{if } M \text{ then } N_1 \text{ else } N_2 : P\{M\}
\end{align*}
\]
Recall that dependent elimination for booleans amounts to

\[
\begin{align*}
\Gamma \vdash M : \mathbb{B} & \quad \Gamma \vdash N_1 : P\{\text{true}\} & \quad \Gamma \vdash N_2 : P\{\text{false}\} \\
\Gamma \vdash \text{if } M \text{ then } N_1 \text{ else } N_2 : P\{M\}
\end{align*}
\]

We proposed a generic restriction for effectful CBN dependent elimination.

\[P \text{ must be linear } (\cong \text{CBV / algebra hom.})\]

- Generalizes Krivine’s storage operators
- If you weren’t at my Geocal-LAC talk, *tant pis pour vous*
- Towards a Linear Dependent \{Big Data, Machine Learning, IoT\}
Shameless Propaganda
Part I

An extension of CIC rooted in Shadok wisdom.

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

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
Part I

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 just say this was not possible???
Part I

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 just say this was not possible???
The exceptional type theory extends vanilla CIC with

\[
\begin{align*}
E & : \Box \\
\text{raise} & : \Pi A : \Box. E \rightarrow A
\end{align*}
\]
The Exceptional Type Theory: Overview

The exceptional type theory extends vanilla CIC with

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

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

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

where \( P : T \rightarrow \Box \).
The Exceptional Type Theory: Overview

The exceptional type theory extends vanilla CIC with

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

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

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

where \( P : \mathcal{I} \rightarrow \Box \).

Remark that in call-by-name, if \( M : A \rightarrow B \), 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**.

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

\[
\begin{align*}
\text{B}_{\text{rect}} : & \quad \Pi P : B \to \Box. \\
P \ \text{true} \to & \\
P \ \text{false} \to & \\
\Pi b : B. \ P \ b
\end{align*}
\]

where

\[
\begin{align*}
catch_B \ P \ pt \ pf \ pe \ \text{true} & \equiv \ pt \\
catch_B \ P \ pt \ pf \ pe \ \text{false} & \equiv \ pf \\
catch_B \ P \ pt \ pf \ pe \ (\text{raise} \ B \ e) & \equiv \ pe \ e
\end{align*}
\]
It’s not just randomly coming up with syntax though.
Mot d’Ordre: A Model

It’s not just randomly coming up with syntax though.

- We want a justification for what we are doing
- What about normalization? Subject reduction? Other nice properties?
It’s not just randomly coming up with syntax though.

- We want a justification for what we are doing
- What about normalization? Subject reduction? Other nice properties?
- ... that’s called a model.

We want a **model of** the exceptional type theory!
Kardashian Functors, Anyone?

Semantics of CIC has a fame of being horribly complex.
Kardashian Functors, Anyone?

Semantics of CIC has a fame of being horribly complex.

I won’t lie: it is. But part of this fame is nonetheless due to its models.
Semantics of CIC has a fame of being horribly complex.

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

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

- **Pro:** Sets!
- **Con:** Sets!
Semantics of CIC has a fame of being horribly complex.

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

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

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

**Realizability** models: construct programs that respect properties.

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

Semantics of CIC has a fame of being horribly complex.

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

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

- Pro: Sets!
- Con: Sets!

**Realizability** models: construct programs that respect properties.

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

**Categorical** models: abstract description of type theory.

- Pro: Abstract, subsumes the two former ones.
- Con: Realizability + very low level, gazillion variants, intrinsically typed, static.
Instead, let’s look at what Curry-Howard provides in simpler settings.

Logical Interpretations $\Leftrightarrow$ Program Translations
Curry-Howard Orthodoxy

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

Logical Interpretations ⇔ Program Translations

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

- A type transformer $T$, two combinators, a few equations
- Interpret mechanically effectful programs (e.g. in Haskell)
Instead, let’s look at what Curry-Howard provides in simpler settings.

Logical Interpretations $\Leftrightarrow$ Program Translations

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

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

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

- Double-negation $\Rightarrow$ classical logic (callcc)
- Friedman’s trick $\Rightarrow$ Markov’s rule (exceptions)
- Forcing $\Rightarrow$ $\neg$CH (global monotonous cell)
Let us do the same thing with CIC: build syntactic models.
Syntactic Models

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

**Step 0:** Fix a theory \( T := \text{CIC} \).
Let us do the same thing with CIC: build **syntactic models**.

**Step 0:** Fix a theory $\mathcal{T} := \text{CIC}$.

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

$$\vdash_{\mathcal{T}} M : A \quad \text{implies} \quad \vdash_{\text{CIC}} \llbracket M \rrbracket : \llbracket A \rrbracket$$
Let us do the same thing with CIC: build **syntactic models**.

**Step 0:** Fix a theory $\mathcal{T} := \text{CIC}$.

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

$$\vdash_{\mathcal{T}} M : A \quad \text{implies} \quad \vdash_{\text{CIC}} [M] : [A]$$

**Step 2:** Flip views and actually pose

$$\vdash_{\mathcal{T}} M : A \quad \triangleq \quad \vdash_{\text{CIC}} [M] : [A]$$
Syntactic Models

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

**Step 0:** Fix a theory $\mathcal{T} := \text{CIC}$.

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

$$\vdash_{\mathcal{T}} M : A \quad \text{implies} \quad \vdash_{\text{CIC}} [M] : [A]$$

**Step 2:** Flip views and actually pose

$$\vdash_{\mathcal{T}} M : A \quad \triangleq \quad \vdash_{\text{CIC}} [M] : [A]$$

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

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

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

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

Every exceptional type comes with its own implementation of failure!
Intuition: \[ \vdash_{\mathcal{E}} A : \Box \leadsto \vdash_{\text{CIC}} [A]: \Sigma A : \Box. \mathbb{E} \to A. \]

Every exceptional type comes with its own implementation of failure!

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

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

Intuition: \( \vdash_{\mathcal{T}_E} A : \square \leadsto \vdash_{\text{CIC}} [A] : \Sigma A : \square . \mathbb{E} \to A. \)

Every exceptional type comes with its own implementation of failure!

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

\[
\begin{align*}
[\Pi x : A. B] & \equiv \Pi x : [A]. [B] \\
[\Pi x : A. B]_{\emptyset} e & \equiv \lambda x : [A]. [B]_{\emptyset} e \\
[x] & \equiv x \\
[M N] & \equiv [M] [N] \\
[\lambda x : A. M] & \equiv \lambda x : [A]. [M]
\end{align*}
\]

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.

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

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

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

It is straightforward to implement the failure operation.

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

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

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

Computational rules trivially hold!

\[
\begin{align*}
[\text{raise} \ (\Pi x : A. B) \ e] & \equiv [\lambda x : A. \text{raise} \ B \ e] \\
\pi_2 \ ((\Pi x : [A]. [B]), (\lambda (e : E) (x : [A]). \pi_2 [B] e)) \ [e] & \equiv \lambda x : [A]. \pi_2 [B] [e]
\end{align*}
\]
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 $[B]_\emptyset : E \to [B]$?

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

... but that would not play well with computation, e.g. catch.
The Exceptional Implementation, Positive case

The really interesting case is the inductive part of CIC.

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

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

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

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

\[
\text{Inductive } [B] := \text{[true]}: [B] \mid \text{[false]}: [B] \mid B_{\emptyset}: E \to [B]
\]
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 \( B_\emptyset e \), \text{reraise } e \text{ 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

😖 Ah, yeah, and also, the theory is inconsistent. It suffices to raise an exception to inhabit any type.
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
- 😞 Ah, yeah, and also, the theory is inconsistent.

It suffices to raise an exception to inhabit any type.
An Impure Dependently-typed Programming Language

Do you whine about the fact that OCaml is logically inconsistent?
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 $e : \mathcal{E}$. 
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!
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)

\[
\text{If } \vdash \tau_E M : \bot, \text{ then } M \equiv \text{raise} \bot e \text{ for some } e : \mathbb{E}.
\]

A Safe Target Framework

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

A Safe Target Framework

You can still use the CIC target to prove properties about \( \text{TE} \) 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

\[
[\neg\neg A] \cong ([A] \to \bot) \to \bot
\]
Informercial — Did You Know?

The exceptional translation is just a principled Friedman’s $\Lambda$-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 $\Lambda$-translation!

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

Informative double-negation

$$\neg\neg A \equiv (\neg A \to E) \to E$$

First-order purification

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

Friedman’s Trick in CIC

If $P$ and $Q$ are $\Sigma^0_1$ types, $\vdash_{CIC} \Pi_p : P. \neg\neg Q$ implies $\vdash_{CIC} \Pi_p : P. Q.$
Part II

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!
The Curry-Howard-Shadok Correspondence

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

What? That’s just logical relations.

Come on. That’s intuitionistic realizability.

Fools! That’s parametricity.

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

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

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

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

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 $M ⊩ A$ if $M$ is valid at $A$.

$$f ⊩ A → B \equiv \forall x : [A]. \ x ⊩ A → f x ⊩ B$$

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.
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.
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

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

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

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

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

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

**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 $M \rightsquigarrow^* \bar{n} \in \mathbb{N}$.

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

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

Intuitively,

- raising uncaught exceptions is forbidden in \( \mathcal{T}_E^p \)
\( \mathcal{T}_E^p \) 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!
$\tau_E^p$ is not a conservative extension of CIC.

Intuitively,

- raising uncaught exceptions is forbidden in $\tau_E^p$
- ... but you can still raise them locally
- ... as long as you prove they don’t escape!

$\tau_E$ is the unsafe Coq fragment, and $\tau_E^p$ 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>Programming language</td>
<td>HA or HA$^\omega$</td>
<td>CIC</td>
</tr>
<tr>
<td>Logical meta-theory</td>
<td>System T</td>
<td>$\mathcal{T}_E$ (“unsafe Coq”)</td>
</tr>
<tr>
<td></td>
<td>HA$^\omega$</td>
<td>CIC</td>
</tr>
</tbody>
</table>
### Explaining the Analogy

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

Kreisel realizability extends arithmetic with essentially two principles.

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

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

Not much to say here.

In Kreisel realizability, \( \text{AC}_N \) is a consequence of canonicity of System T.
\[ AC_N : (\forall n : \mathbb{N} \cdot \exists m : \mathbb{N} \cdot P(m, n)) \rightarrow \exists f : \mathbb{N} \rightarrow \mathbb{N} \cdot \forall n : \mathbb{N} \cdot P(n, f \ n) \]

Not much to say here.

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

In \( T^p_E \), \( AC_N \) 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 $T^p_E$, 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 : (¬A → ∃n : N. P n) → ∃n : N. ¬A → P n

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

Kreisel’s bug

Every type of realizers is inhabited. In particular, ⌈⊥⌉_{KR} ≡ 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 \textbf{bug}.

\textbf{Kreisel’s bug}

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

The realizer of IP critically relies on that!

Assuming System T had an empty type \(0\), and setting \([\bot]_{\text{KR}} \equiv 0\)

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

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

$T_E^p$ validates IP, owing to the fact that in $T_E$, every type is inhabited.

Proof (sketch).

In $T_E$, build a term $ip : IP$

- Given $f : \neg A \rightarrow \Sigma n : \mathbb{N}. P n$, apply it to raise $\neg A$.
- 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 $T_E^p$. 

\[
IP : (\neg A \rightarrow \exists n : \mathbb{N}. P n) \rightarrow \exists n : \mathbb{N}. \neg A \rightarrow P n
\]
Recall Markov’s principle:

\[
\Pi P : \mathbb{N} \rightarrow \mathbb{B}. \neg (\Sigma n : \mathbb{N}. P n = \text{true}) \rightarrow \Sigma n : \mathbb{N}. P n = \text{true} \quad \text{(MP)}
\]
Another Result for Free

Recall Markov’s principle:

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

Kreisel’s Razor

Pick two out of three: \{canonicity, IP, MP\}.
Recall Markov’s principle:

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

**Kreisel’s Razor**

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

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

Together with canonicity, this solves the halting problem.
Recall Markov’s principle:

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

(KP)

Kreisel’s Razor

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

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

Together with canonicity, this solves the halting problem.

Corollary

\[ \vdash \text{\textit{MP} and thus} \vdash_{\text{CIC}} \text{MP}. \]

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

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

\[
\vdash_{\mathcal{T}^p_{\text{IE}}} \Pi i : 1. i = \texttt{tt} \quad \text{and} \quad \vdash_{\mathcal{T}^p_{\text{IE}}} (\lambda i : 1. i) \neq (\lambda i : 1. \texttt{tt})
\]
Another interesting consequence that is similar to what happens in KR.

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

$$\vdash_{\mathcal{T}_E^p} \Pi i : 1. i = tt \quad \text{and} \quad \vdash_{\mathcal{T}_E^p} (\lambda i : 1. i) \neq (\lambda i : 1. tt)$$

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

You cannot build them, but they exist as phantom arguments.
What kind of similar horrors can we do in $\mathcal{T}_E^p$?

- I don’t know!
- But there are probably lessons to be taken from realizability
- I’m probably pissing off both HoTT and PRL zealots by now
Get You A Larger Coq, Today!

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!
$\mathcal{T}_E$, 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 $A$-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, $\neg$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 \( \lambda \)-translation

\( \mathcal{T}^p_E \), 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

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 $A$-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, $\neg$funext, disproves MP

Both of them justified by purely syntactical means!

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

P.-M. Pédrot (MPI-SWS)  Failure is Not an Option  22/02/2018  41 / 44
• $\mathcal{T}_E$ looks like a good intermediate language for model building
• The Calculus of Shadok Constructions
• Potential applications to Gradual Typing?
• Syntactic models are super cool! Let’s write more!
C'EST
TOUT
POUR
AUJOURD'HUI
It seems you need to have a name starting with K to name a realizability.

Kleene
Kreisel
Krivine