RETTR, a Reasonably Exceptional Type Theory

Pierre-Marie Pédrot$^1$, Nicolas Tabareau$^1$, Hans Fehrmann$^2$, Éric Tanter$^{1, 2}$

$^1$INRIA, $^2$University of Chile

ICFP 2019
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
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
¿ CIC, a very powerful **functional programming language**?
¿ CIC, a very powerful **functional** programming language? 

... as long as you can live in purely functional fantasyland

- No native effects (not even non-termination!)
- Haskell monadic style awkward with dependent types
- What is even the point of using Coq then?
¿ CIC, a very powerful functional programming language? 

... as long as you can live in purely functional fantasyland

- No native effects (not even non-termination!)
- Haskell monadic style awkward with dependent types
- What is even the point of using Coq then?

CIC, a not so great effectful programming language 😞
We have been working on extending CIC with **side-effects**.
We have been working on extending CIC with **side-effects**.

- To program more, but also to **prove** more
- Justification via compilation (more on that soon)
- A lot of interesting stuff to say but time is pressing
We have been working on extending CIC with side-effects.

- To program more, but also to prove more
- Justification via compilation (more on that soon)
- A lot of interesting stuff to say but time is pressing

Effect *du jour*

We will concentrate today on only one particular, simple effect.
We have been working on extending CIC with side-effects.

- To program more, but also to prove more
- Justification via compilation (more on that soon)
- A lot of interesting stuff to say but time is pressing

Effect *du jour*

We will concentrate today on only one particular, simple effect.

EXCEPTIONS!
ExTT, an extension of CIC with exceptions.

- Add a failure mechanism to CIC
- Fully computational call-by-name exceptions
- Contains the whole of CIC (including krazy dependent stuff)
- Compiled away to vanilla CIC (so-called syntactic model)
ExTT, an extension of CIC with exceptions.

- Add a failure mechanism to CIC
- Fully computational call-by-name exceptions
- Contains the whole of CIC (including krazy dependent stuff)
- Compiled away to vanilla CIC (so-called syntactic model)

Let’s have a look at ExTT!
The Exceptional Type Theory: Overview

ExTT extends CIC with an \textbf{exception-raising} primitive (ITT: no payload).

\begin{align*}
\text{raise} & : \prod A : \Box. A \\
\text{raise} (\Pi x : A. B) & \equiv \lambda x : A. \text{raise } B \\
\text{match} (\text{raise } I) \text{ ret } P \text{ with } \bar{p} & \equiv \text{raise } (P \text{ (raise } I))
\end{align*}
ExTT extends CIC with an exception-raising primitive (ITT: no payload).

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

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

\[
\text{match} (\text{raise} I) \text{ ret } P \text{ with } \vec{p} \equiv \text{raise} (P (\text{raise} I))
\]

They can be caught on inductive types via a generalization of eliminators.

\[
\text{B}_{\text{rec}} : \ \Pi P : \mathbb{B} \to \Box.
\]

\[
P \text{ true } \to
\]

\[
P \text{ false } \to \sim
\]

\[
\Pi b : \mathbb{B}. \ P \ b
\]

\[
\text{catch}_{\mathbb{B}} : \ \Pi P : \mathbb{B} \to \Box.
\]

\[
P \text{ true } \to
\]

\[
P \text{ false } \to
\]

\[
P (\text{raise} \mathbb{B}) \to
\]

\[
\Pi b : \mathbb{B}. \ P \ b
\]

\[
\text{catch}_{\mathbb{B}} P \ pt \ pf \ pe \text{ true} \equiv pt
\]

\[
\text{catch}_{\mathbb{B}} P \ pt \ pf \ pe \text{ false} \equiv pf
\]

\[
\text{catch}_{\mathbb{B}} P \ pt \ pf \ pe (\text{raise} \mathbb{B}) \equiv pe
Exception is the Rule

While a fairly simple effect, exceptions are already super useful!
Exception is the Rule

While a fairly simple effect, exceptions are already super useful!

Dead code stays so.
« Come on Coq, I know this branch cannot occur! »
Use raise.
While a fairly simple effect, exceptions are already super useful!

**Dead code stays so.**

« Come on Coq, I know this branch cannot occur! »

Use `raise`.

**Post-hoc reasoning.**

« Why do I need to thread decidable hypotheses everywhere? »

Use `raise`.
While a fairly simple effect, exceptions are already super useful!

**Dead code stays so.**

« Come on Coq, I know this branch cannot occur! »

Use raise.

**Post-hoc reasoning.**

« Why do I need to thread decidable hypotheses everywhere? »

Use raise.

**Failure as a default.**

« Why on earth do I have to return an option? »

Use raise.

Typical problems from the wild: mathcomp, hs-to-coq...
How do we prove that ExTT makes any sense?

- We want a justification for what we are doing
- What about normalization? Subject reduction? Other nice properties?
How do we prove that ExTT makes any sense?

- We want a justification for what we are doing
- What about normalization? Subject reduction? Other nice properties?

We want a **model of** the exceptional type theory!
How do we prove that ExTT makes any sense?

- We want a justification for what we are doing
- What about normalization? Subject reduction? Other nice properties?

We want a **model of** the exceptional type theory!

"compilation"

« CIC, the LLVM of Type Theory »
The Exceptional Implementation (sketch)

A Truly Simple Model!

$$
\begin{align*}
\vdash_{\text{ExTT}} & A : \square \quad \leadsto \quad \vdash_{\text{CIC}} [A] : \square & + & \vdash_{\text{CIC}} [A]_\emptyset : [A] \\
\vdash_{\text{ExTT}} & M : A \quad \leadsto \quad \vdash_{\text{CIC}} [M] : [A]
\end{align*}$$

Every exceptional type comes with its own implementation of failure!
The Exceptional Implementation (sketch)

A Truly Simple Model!

\[ \vdash_{\text{ExTT}} A : \square \rightsquigarrow \vdash_{\text{CIC}} [A] : \square + \vdash_{\text{CIC}} [A]\emptyset : [A] \]

\[ \vdash_{\text{ExTT}} M : A \rightsquigarrow \vdash_{\text{CIC}} [M] : [A] \]

Every exceptional type comes with its own implementation of failure!

\[ [\square] := \Sigma A : \square. A \]
\[ [\Pi x : A. B] := \Pi x : [A]. [B] \]
The Exceptional Implementation (sketch)

A Truly Simple Model!

\[ \vdash_{\text{ExTT}} A : \square \leadsto \vdash_{\text{CIC}} \lbrack A \rbrack : \square + \vdash_{\text{CIC}} \lbrack A \rbrack_{\emptyset} : \lbrack A \rbrack \]
\[ \vdash_{\text{ExTT}} M : A \leadsto \vdash_{\text{CIC}} \lbrack M \rbrack : \lbrack A \rbrack \]

Every exceptional type comes with its own implementation of failure!

\[
\begin{align*}
\lbrack \square \rbrack & := \Sigma A : \square. A \\
\lbrack \Pi x : A. B \rbrack & := \Pi x : \lbrack A \rbrack. \lbrack B \rbrack \\
\lbrack \square \rbrack_{\emptyset} & := \ldots \\
\lbrack \Pi x : A. B \rbrack_{\emptyset} & := \lambda x : \lbrack A \rbrack. \lbrack B \rbrack_{\emptyset}
\end{align*}
\]
The Exceptional Implementation (sketch)

A Truly Simple Model!

\[\vdash_{\text{ExTT}} A : \Box \leadsto \vdash_{\text{CIC}} [A] : \Box + \vdash_{\text{CIC}} [A] \emptyset : [A]\]

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

Every exceptional type comes with its own implementation of failure!

\[
\begin{align*}
[\Box] & := \Sigma A : \Box. A \\
[\Pi x : A. B] & := \Pi x : [A]. [B] \\
[\Box] \emptyset & := \ldots \\
[\Pi x : A. B] \emptyset & := \lambda x : [A]. [B] \emptyset \\
[M] & := \ldots \\
[\text{raise } A] & := [A] \emptyset
\end{align*}
\]
The Exceptional Implementation, Positive case

Add an **error case** to every inductive type!

\[
\text{Inductive } \mathbf{B} := \text{[true]} : \mathbf{B} \mid \text{[false]} : \mathbf{B} \mid \mathbf{B} \varnothing : \mathbf{B}
\]
Add an \textbf{error case} to every inductive type!

\begin{align*}
\text{Inductive } & [\mathbb{B}] := [\text{true}] : [\mathbb{B}] \mid [\text{false}] : [\mathbb{B}] \mid \mathbb{B}_\emptyset : [\mathbb{B}]
\end{align*}

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

\[
[\Pi P : \mathbb{B} \rightarrow \emptyset. P \text{ true} \rightarrow P \text{ false} \rightarrow \Pi b : \mathbb{B}. P b]
\equiv
\Pi P : [\mathbb{B}] \rightarrow [\emptyset]. P [\text{true}] \rightarrow P [\text{false}] \rightarrow \Pi b : [\mathbb{B}]. P b
\]

\begin{itemize}
  \item If \( b \) is \([\text{true}]\), use first hypothesis
  \item If \( b \) is \([\text{false}]\), use second hypothesis
  \item If \( b \) is an error \( \mathbb{B}_\emptyset \), \textbf{reraise} using \( [P b]_\emptyset \)
\end{itemize}
Where is the fish?

Theorem ExTT \textit{contains} CIC...
Theorem

ExTT *contains* CIC... *but it also proves everything.* 😊 *(Use raise!)*
Where is the fish?

Theorem

ExTT contains CIC...but it also proves everything. 😓 (Use raise!)

An Impure Dependently-typed Programming Language

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

Theorem

ExTT contains CIC... *but it also proves everything.* 😖 (Use raise!)

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* $\vdash_{\text{E}x\text{TT}} M : \bot$, *then* $M \equiv \text{raise} \bot$. 
In ESOP 2018 we described pExTT, a **consistent** restriction of ExTT.

- Variant of Bernardy-Lasson style parametricity (syntactic model)
- Toplevel exceptions forbidden, but can still be raised locally (meh)

\[
\text{CIC} \not\subseteq \text{pExTT} \not\subseteq \text{ExTT}
\]
With Great Effects Come Great Responsibility

In ESOP 2018 we described pExTT, a **consistent** restriction of ExTT.

- Variant of Bernardy-Lasson style parametricity (syntactic model)
- Toplevel exceptions forbidden, but can still be raised locally (meh)

\[
\text{CIC} \not\subset p\text{ExTT} \subset \text{ExTT}
\]

Now we have a dilemma!

<table>
<thead>
<tr>
<th>ExTT</th>
<th>pExTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>☹️ Inconsistent</td>
<td>☺️ Consistent</td>
</tr>
<tr>
<td>☺️ Unrestricted use of exceptions</td>
<td>☒ Exceptions virtually unusable</td>
</tr>
<tr>
<td>☻️ Good for programming</td>
<td>☴ Strange logical properties</td>
</tr>
</tbody>
</table>
Tired to have to make a choice? We have the answer!
Tired to have to make a choice? We have the answer!

\[
\text{CIC } \not\subseteq \text{ pExTT } \not\subseteq \text{ ExTT}
\]
Tired to have to make a choice? We have the answer!

with not one, not two, but three universe hierarchies
Tired to have to make a choice? We have the answer!

with not one, not two, but three universe hierarchies

- **Pure Layer**
  - \( \Box^p_i \sim \text{CIC} \)
  - Consistent
  - No exceptions
  - For proving

\[ \text{CIC} \subsetneq \text{pExTT} \subsetneq \text{ExTT} \]
Tired to have to make a choice? We have the answer!

with not one, not two, but **three** universe hierarchies

- **Pure Layer**
  - $\square^p_i \sim \text{CIC}$
  - Consistent
  - No exceptions
  - For proving

- **Exceptional Layer**
  - $\square^e_i \sim \text{ExTT}$
  - Inconsistent
  - Full exceptions
  - For programming
Tired to have to make a choice? We have the answer!

with not one, not two, but three universe hierarchies

<table>
<thead>
<tr>
<th>Pure Layer</th>
<th>Exceptional Layer</th>
<th>Mediating Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\square^p_i \sim \text{CIC}$</td>
<td>$\square^e_i \sim \text{ExTT}$</td>
<td>$\square^m_i \sim \text{pExTT}$</td>
</tr>
<tr>
<td>▶ Consistent</td>
<td>▶ Inconsistent</td>
<td>▶ Consistent</td>
</tr>
<tr>
<td>▶ No exceptions</td>
<td>▶ Full exceptions</td>
<td>▶ Local exceptions</td>
</tr>
<tr>
<td>▶ For proving</td>
<td>▶ For programming</td>
<td>▶ For communication</td>
</tr>
</tbody>
</table>
At the Crossroads

Every hierarchy in isolation behaves as a variant of CIC

\[ □^p_i \sim \text{CIC} \quad □^e_i \sim \text{ExTT} \quad □^m_i \sim \text{pExTT} \]
Every hierarchy in isolation behaves as a variant of CIC

\[ \square_i^p \sim \text{CIC} \quad \square_i^e \sim \text{ExTT} \quad \square_i^m \sim \text{pExTT} \]

“Write programs in \( \square^e \),
Prove them in \( \square^m \) or \( \square^p \).”
At the Crossroads

Every hierarchy in isolation behaves as a variant of CIC

\[ \square^p_i \sim \text{CIC} \quad \square^e_i \sim \text{ExTT} \quad \square^m_i \sim \text{pExTT} \]

“Write programs in \( \square^e \),
Prove them in \( \square^m \) or \( \square^p \).”

The expressivity of RETT lies in the interaction between hierarchies

\[
\Gamma \vdash A : \square^\alpha_i \\
\Gamma, x : A \vdash B : \square^\beta_j \\
\alpha, \beta \in \{\text{p, e, m}\}
\]

\[
\Gamma \vdash \Pi(x : A). B : \square^\beta_{i \lor j}
\]
Eliminating Between Hierarchies

Eliminating inductive types is even more interesting

CIC
\[ \mathbb{B}_{\text{rec}} : \Pi P : \mathbb{B} \rightarrow \Box. P \text{ true} \rightarrow P \text{ false} \rightarrow \Pi b : \mathbb{B}. P b \]

ExTT
\[ \mathbb{B}_{\text{catch}} : \Pi P : \mathbb{B} \rightarrow \Box. P \text{ true} \rightarrow P \text{ false} \rightarrow P (\text{raise } \mathbb{B}) \rightarrow \Pi b : \mathbb{B}. P b \]
Eliminating between hierarchies

Eliminating inductive types is even more interesting

<table>
<thead>
<tr>
<th>CIC</th>
<th>$\mathbb{B}_{\text{rec}} : \Pi P : \mathbb{B} \to \square. P \text{ true} \to P \text{ false} \to \Pi b : \mathbb{B}. P \ b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExTT</td>
<td>$\mathbb{B}_{\text{catch}} : \Pi P : \mathbb{B} \to \square. P \text{ true} \to P \text{ false} \to P (\text{raise } \mathbb{B}) \to \Pi b : \mathbb{B}. P \ b$</td>
</tr>
</tbody>
</table>

Depending on the hierarchy of $\mathbb{B}$ and $P$ we get different eliminators!

<table>
<thead>
<tr>
<th>$P : \mathbb{B} \to \square^\beta$</th>
<th>$\square^e$</th>
<th>$\square^m$</th>
<th>$\square^p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\square^e$</td>
<td>rec/catch</td>
<td>catch</td>
<td>catch</td>
</tr>
<tr>
<td>$\mathbb{B} : \square^\alpha$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\square^m$</td>
<td>rec</td>
<td>rec</td>
<td>–</td>
</tr>
<tr>
<td>$\square^p$</td>
<td>rec</td>
<td>rec</td>
<td>rec</td>
</tr>
</tbody>
</table>

- catch does not make sense when source is consistent
- catch is mandatory when eliminating from inconsistent to consistent
- reminiscent of the singleton elimination restriction in CIC.
And Much More

We also have modalities for better interoperability

\[ \{ - \}^\alpha_\beta : \square^\alpha \to \square^\beta \]
\[ \iota^\alpha_\beta : \Pi(A : \square^\alpha). A \to \{ A \}^\alpha_\beta \]

... as well as an internal purity predicate

\[ \mathcal{P} : \Pi(A : \square^m). \{ A \}^m_\epsilon \to \square^m \]
\[ \Sigma(x : \{ A \}^m_\epsilon). \mathcal{P} A x \equiv A \]

Main interest of \( \square^m \) over \( \square^p \).

(A lot to say, but I don’t have time.)
We implemented RETT as a POC Coq plugin.

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

- Allows to add exceptions to Coq just today.
- Piggybacks on the Prop/Type segregation (hack hack hack)
- Compile RETT on the fly.
- Not really practical though, should this go into the kernel?
• Actually provide RETT first class in Coq?
• Use it for programming for realz?
• Potential applications to Gradual Typing?
• One hierarchy = one effect?
- Actually provide RETT first class in Coq?
- Use it for programming for realz?
- Potential applications to Gradual Typing?
- One hierarchy = one effect?
If You Were Sleeping During The Talk

RETT, a 3-in-1 type theory!

1. An inconsistent dependently-typed effectful programming language
2. A consistent dependently-typed proof language
3. A consistent dependently-typed mediating language

Smoothly interacting together!

All of this justified by purely syntactical means!

Implemented in your favourite proof assistant!