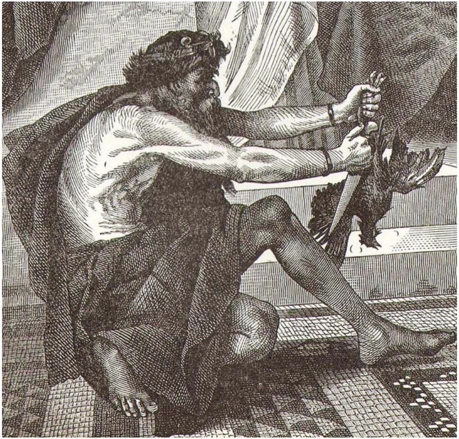
Dans les Boyaux de mon Noyau

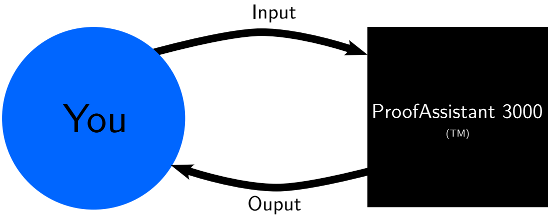


Pierre-Marie Pédrot

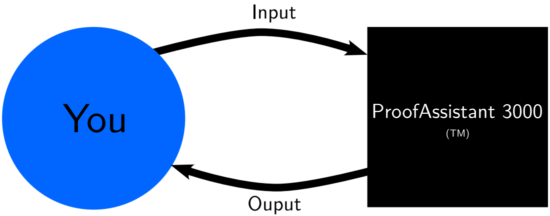
(Gallinette, INRIA)

JNIM'24

Des joies† de l'Assistanat à la Preuve



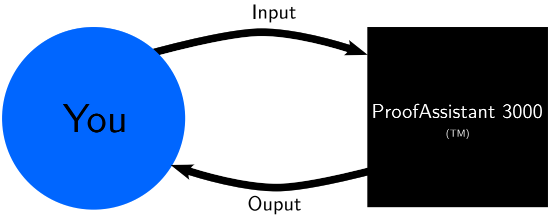
Des joies† de l'Assistanat à la Preuve



Laborious interactive input:

- Theorem statements and object definitions

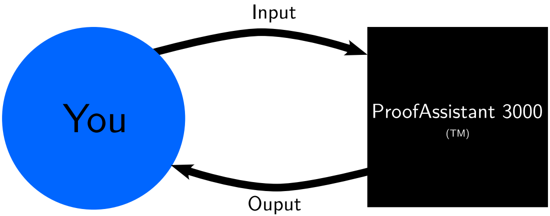
Des joies† de l'Assistanat à la Preuve



Laborious interactive input:

- Theorem statements and object definitions
- Proof arguments e.g. tactics

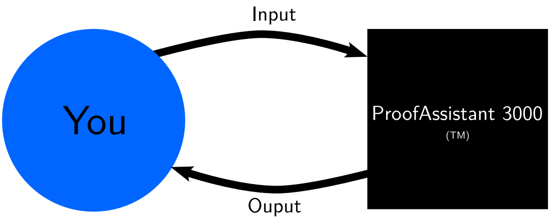
Des joies† de l'Assistanat à la Preuve



Laborious interactive input:

- Theorem statements and object definitions
- Proof arguments e.g. tactics
- Copious amounts of insults

Des joies† de l'Assistanat à la Preuve



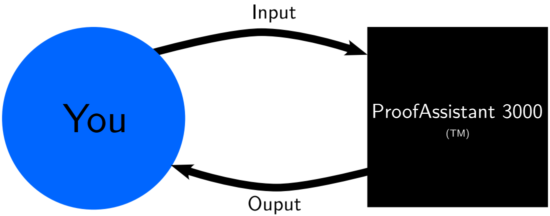
Laborious interactive input:

- Theorem statements and object definitions
- Proof arguments e.g. tactics
- Copious amounts of insults

Cold mechanical output:

- *Good!* Here's what remains to do.
- `Nope.`
- (undecypherable dump of some low-level data)

Des joies† de l'Assistanat à la Preuve



Laborious interactive input:

- Theorem statements and object definitions
- Proof arguments e.g. tactics
- Copious amounts of insults

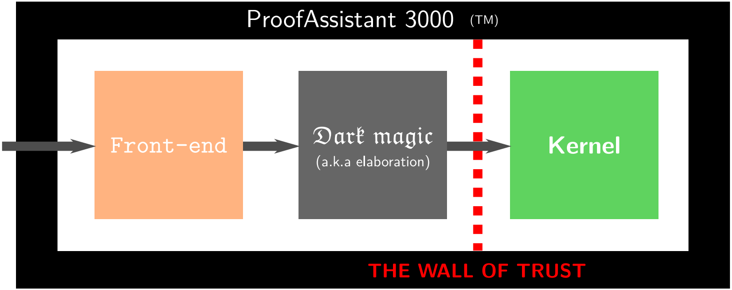
Cold mechanical output:

- *Good!* Here's what remains to do.
- `Nope.`
- (undecypherable dump of some low-level data)

† Your mileage may vary.

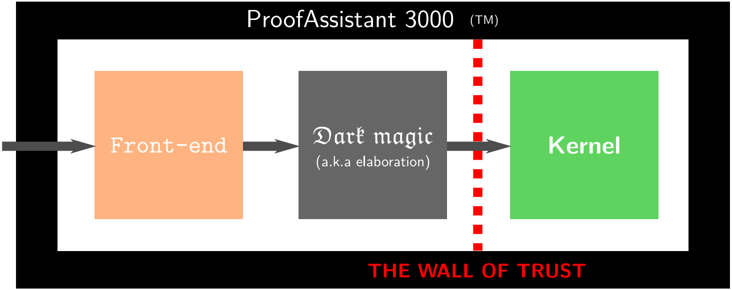
Elle C'est Heffe

A well-known design: The LCF Model



Elle C'est Heffe

A well-known design: The LCF Model



- Clearly delineated Trusted Code Base
- All fancy stuff is outside the TCB
- Soundness is reduced to a *small* hopefully understandable kernel

Two standard kind of kernels in the wild

Two standard kind of kernels in the wild

First kind: *canal historique* still living in the HOL family

- Kernel is a trusted minimal API of HOL tactics
- Proofs not recorded, soundness ensured by the metalanguage

Noyal-$\lambda\mu$zillac

Two standard kind of kernels in the wild

First kind: *canal historique* still living in the HOL family

- Kernel is a trusted minimal API of HOL tactics
- Proofs not recorded, soundness ensured by the metalanguage

Second kind: when proofs matter as e.g. in dependent type theories

- Kernel is a type-checker
- Proofs are well-typed terms

Two standard kind of kernels in the wild

First kind: *canal historique* still living in the HOL family

- Kernel is a trusted minimal API of HOL tactics
- Proofs not recorded, soundness ensured by the metalanguage

Second kind: when proofs matter as e.g. in dependent type theories

- Kernel is a type-checker
- Proofs are well-typed terms

Official Position

In this talk, we care about dependent type theories

« Constructions dans un monde qui bouge »

CIC, the calculus of inductive constructions.

CIC, the calculus of inductive constructions.

- A powerful dependent type theory
- Programming language or logical foundation?
- The idealized basis of two famous proof assistants

« Constructions dans un monde qui bouge »

CIC, the calculus of inductive constructions.

- A powerful dependent type theory
- Programming language or logical foundation?
- The idealized basis of two famous proof assistants

Full Disclaimer

I am a core Coq developer.

Un théoricien des types c'est un système†

Full Disclaimer

I am an **opinionated** core Coq developer.

Un théoricien des types c'est un système†

Full Disclaimer

I am an **opinionated** core Coq developer.

Thankfully, most of what I am saying should apply to other proof assistants based on dependent type theories.

- Kernel-wise, Coq and Lean are very close.
- I can still rant forever about some subtle differences in design
- Hint: Coq does it right (most of the time)

Un théoricien des types c'est un système†

Full Disclaimer

I am an **opinionated** core Coq developer.

Thankfully, most of what I am saying should apply to other proof assistants based on dependent type theories.

- Kernel-wise, Coq and Lean are very close.
- I can still rant forever about some subtle differences in design
- Hint: Coq does it right (most of the time)

Some of what I will say even applies to Agda

Agda

… even if Agda has no separate kernel.

Un théoricien des types c'est un système†

Full Disclaimer

I am an **opinionated** core Coq developer.

Thankfully, most of what I am saying should apply to other proof assistants based on dependent type theories.

- Kernel-wise, Coq and Lean are very close.
- I can still rant forever about some subtle differences in design
- Hint: Coq does it right (most of the time)

Some of what I will say even applies to Agda

... even if Agda has no separate kernel.

† — ***Deux c'est une école. Trois, c'est un fork.***

CIC: Programming language or logical foundation?

CIC: Programming language or logical foundation?

Both!

CIC: Programming language or logical foundation?

Both!

Level 0: $\vdash (\lambda(x : A).\, x) : A \to A$

- the identity function on A?
- the canonical proof that A implies A?

CIC: Programming language or logical foundation?

Both!

Level 0: $\vdash (\lambda(x : A).\, x) : A \to A$

- the identity function on A?
- the canonical proof that A implies A?

Level 1: $\vdash M : \Pi(m : \mathbb{N}).\, \Sigma(p : \mathbb{N}).(m = 2p) + (m = 2p + 1)$

- an implementation of division by 2?
- a proof that division by 2 exists?

CIC: Programming language or logical foundation?

Both!

Level 0: $\vdash (\lambda(x : A).\, x) : A \to A$

- the identity function on A?
- the canonical proof that A implies A?

Level 1: $\vdash M : \Pi(m : \mathbb{N}).\, \Sigma(p : \mathbb{N}).(m = 2p) + (m = 2p + 1)$

- an implementation of division by 2?
- a proof that division by 2 exists?

Level 2: $\vdash M : \Pi(b : \mathbb{B}).\, \mathtt{if}\ b\ \mathtt{then}\ \mathbb{N}\ \mathtt{else}\ (\mathbb{N} \to \mathbb{N})$

CIC: Programming language or logical foundation?

Both!

Level 0: $\vdash (\lambda(x : A).\, x) : A \to A$

- the identity function on A?
- the canonical proof that A implies A?

Level 1: $\vdash M : \Pi(m : \mathbb{N}).\, \Sigma(p : \mathbb{N}).(m = 2p) + (m = 2p + 1)$

- an implementation of division by 2?
- a proof that division by 2 exists?

Level 2: $\vdash M : \Pi(b : \mathbb{B}).\, \texttt{if}\ b\ \texttt{then}\ \mathbb{N}\ \texttt{else}\ (\mathbb{N} \to \mathbb{N})$

Level 42: *(spec of CompCert)*

Un Monisme en Deux Minutes

CIC: Programming language or logical foundation?

Both!

Level 0: $\vdash (\lambda(x : A).\, x) : A \to A$

- the identity function on A?
- the canonical proof that A implies A?

Level 1: $\vdash M : \Pi(m : \mathbb{N}).\, \Sigma(p : \mathbb{N}).(m = 2p) + (m = 2p + 1)$

- an implementation of division by 2?
- a proof that division by 2 exists?

Level 2: $\vdash M : \Pi(b : \mathbb{B}).\, \mathtt{if}\ b\ \mathtt{then}\ \mathbb{N}\ \mathtt{else}\ (\mathbb{N} \to \mathbb{N})$

Level 42: *(spec of CompCert)*

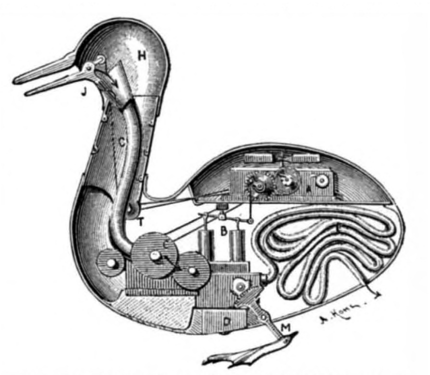
The pinacle of the Curry-Howard correspondence

Réductionnisme de Basse-Cour

The Coq kernel is *just* a CIC type-checker

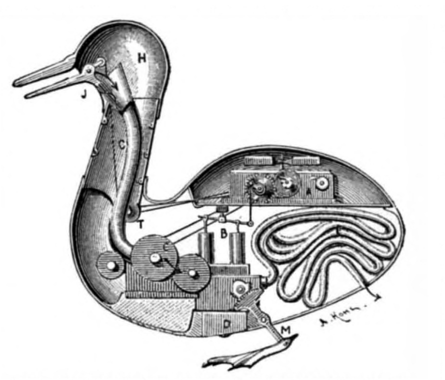
Réductionnisme de Basse-Cour

The Coq kernel is *just* a CIC type-checker



Réductionnisme de Basse-Cour

The Coq kernel is *just* a CIC type-checker



Famous Last Words

"Surely it should be enough to understand CIC to understand the kernel."

Ceci n'est pas un Π

"Surely it should be enough to understand CIC to understand the kernel."

Ceci n'est pas un Π

"Surely it should be enough to understand CIC to understand the kernel."

I Lied

There is no such thing as CIC.

Ceci n'est pas un Π

"Surely it should be enough to understand CIC to understand the kernel."

I Lied

There is no such thing as CIC.

- A bunch of rules spread across dozen of articles spanning decades
- No single source of truth
- A lot of implicit or contradictory stuff
- Folklore / unwritten knowledge

Ceci n'est pas un Π

"Surely it should be enough to understand CIC to understand the kernel."

I Lied

There is no such thing as CIC.

- A bunch of rules spread across dozen of articles spanning decades
- No single source of truth
- A lot of implicit or contradictory stuff
- Folklore / unwritten knowledge

Not better implementation-wise.

- Takes some suspect liberties w.r.t. the spec
- It keeps changing

Ceci n'est pas un Π

"Surely it should be enough to understand CIC to understand the kernel."

I Lied

There is no such thing as CIC.

- A bunch of rules spread across dozen of articles spanning decades
- No single source of truth
- A lot of implicit or contradictory stuff
- Folklore / unwritten knowledge

Not better implementation-wise.

- Takes some suspect liberties w.r.t. the spec
- It keeps changing

Our best bet: the MetaCoq project. But that's not today's topic.

Ora Pro Nobis

“The Non-Existent Type Theory is based upon both logic and faith. We have faith that it is implemented; we logically know that it is does not exist because it is not well-defined.”

Ora Pro Nobis

"The Non-Existent Type Theory is based upon both logic and faith. We have faith that it is implemented; we logically know that it is does not exist because it is not well-defined."

We will pretend that CIC exists in the remainder of the talk.

Persevere Diabolicum

"Surely it should be enough to understand CIC to understand the kernel."

... for some good notion of CIC.

Persevere Diabolicum

"Surely it should be enough to understand CIC to understand the kernel."

... for some good notion of CIC.

Still Wrong

Reality has many asperities.

Persevere Diabolicum

“Surely it should be enough to understand CIC to understand the kernel.”

... for some good notion of CIC.

Still Wrong

Reality has many asperities.

A specification is not an implementation.

The Good Properties of CIC

The Good Properties of CIC

Consistency: The Logician

There is no proof $\vdash M : \bot$.

The Good Properties of CIC

Consistency: The Logician

There is no proof $\vdash M : \bot$.

... but the logic must be as expressive as possible.

The Good Properties of CIC

Consistency: The Logician

There is no proof $\vdash M : \bot$.

... but the logic must be as expressive as possible.

Canonicity: The Programmer

If $\vdash M : \mathbb{N}$ then M evaluates to a numeral.

The Good Properties of CIC

Consistency: The Logician

There is no proof $\vdash M : \bot$.

... but the logic must be as expressive as possible.

Canonicity: The Programmer

If $\vdash M : \mathbb{N}$ then M evaluates to a numeral.

... but I want pointer equality, exceptions and Turing-completeness.

The Good Properties of CIC

Consistency: The Logician

There is no proof $\vdash M : \bot$.

... but the logic must be as expressive as possible.

Canonicity: The Programmer

If $\vdash M : \mathbb{N}$ then M evaluates to a numeral.

... but I want pointer equality, exceptions and Turing-completeness.

Implementability: The Maintainer

Type-checking is decidable.

The Good Properties of CIC

Consistency: The Logician

There is no proof $\vdash M : \bot$.

... but the logic must be as expressive as possible.

Canonicity: The Programmer

If $\vdash M : \mathbb{N}$ then M evaluates to a numeral.

... but I want pointer equality, exceptions and Turing-completeness.

Implementability: The Maintainer

Type-checking is decidable.

... but I am going to add orthogonal features X, Y and Z.

Rapports de force et personnalités multiples

A Type Theory implementation is a delicate equilibrium.

Rapports de force et personnalités multiples

A Type Theory implementation is a delicate equilibrium.

$\rightsquigarrow$ Expanding the logic must preserve computation.

- One cannot just add axioms here and there
- Some constructions are not even axiomatizable (e.g. cubicalTT)

Rapports de force et personnalités multiples

A Type Theory implementation is a delicate equilibrium.

$\rightsquigarrow$ Expanding the logic must preserve computation.

- One cannot just add axioms here and there
- Some constructions are not even axiomatizable (e.g. cubicalTT)

$\rightsquigarrow$ Expanding the programming language must preserve consistency.

- Extremely strong constraints, e.g. functions are total
- A lot of stuff from PLT just doesn't apply

Rapports de force et personnalités multiples

A Type Theory implementation is a delicate equilibrium.

$\rightsquigarrow$ Expanding the logic must preserve computation.

- One cannot just add axioms here and there
- Some constructions are not even axiomatizable (e.g. cubicalTT)

$\rightsquigarrow$ Expanding the programming language must preserve consistency.

- Extremely strong constraints, e.g. functions are total
- A lot of stuff from PLT just doesn't apply

$\rightsquigarrow$ Expanding any of these must keep the implementation tractable.

- Understandable spec / small implementation
- Efficient algorithmics
- Backwards compatibility

Rapports de force et personnalités multiples

A Type Theory implementation is a delicate equilibrium.

$\rightsquigarrow$ Expanding the logic must preserve computation.

- One cannot just add axioms here and there
- Some constructions are not even axiomatizable (e.g. cubicalTT)

$\rightsquigarrow$ Expanding the programming language must preserve consistency.

- Extremely strong constraints, e.g. functions are total
- A lot of stuff from PLT just doesn't apply

$\rightsquigarrow$ Expanding any of these must keep the implementation tractable.

- Understandable spec / small implementation
- Efficient algorithmics
- Backwards compatibility

The logician, programmer and maintainer are often the same individual.

Pardon My French

— Un noyau, c'est comme une andouillette:
ça doit sentir un peu la merde, mais pas trop.

L'œuf vif du sujet

The setting is now pinned down

In this talk we will discuss three interesting components of the Coq kernel.

Conversion

Universes

Guard

All while keeping the andouillette principle in mind!

Conversion

THE MOST IMPORTANT RULE OF CIC

Le changement, c'est maintenant

THE MOST IMPORTANT RULE OF CIC

(If you were snoozing away, now is the time to wake up.)

THE MOST IMPORTANT RULE OF CIC

(If you were snoozing away, now is the time to wake up.)

Meet CONVERSION:

$$\frac{\Gamma \vdash M : B \quad \Gamma \vdash A \equiv B}{\Gamma \vdash M : A}$$

THE MOST IMPORTANT RULE OF CIC

(If you were snoozing away, now is the time to wake up.)

Meet CONVERSION:

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

$$\mathtt{refl}_A : \Pi(x : A).\, x = x \qquad \rightsquigarrow \qquad (\mathtt{refl}_{\mathbb{N}}\ 2) : 1 + 1 = 2$$

THE MOST IMPORTANT RULE OF CIC

(If you were snoozing away, now is the time to wake up.)

Meet CONVERSION:

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

$$\mathtt{refl}_A : \Pi(x : A).\, x = x \qquad \rightsquigarrow \qquad (\mathtt{refl}_{\mathbb{N}}\ 2) : 1 + 1 = 2$$

CONVERSION internalizes computation in the logic

- Not common in usual PL
- Irremediably ties the runtime to the type system
- A landmark of dependent types

Séquent qu'on calcule

$$\Gamma \vdash A \equiv B$$

What is conversion exactly?

Séquent qu'on calcule

$$\Gamma \vdash A \equiv B$$

What is conversion exactly?

Generated by hardwired basic equations on the language e.g.

- β-reduction: $(\lambda(x : A).\, M)\ N \equiv M\{x := N\}$
- pattern-matching reduction on constructors
- constant unfolding

Séquent qu'on calcule

$$\Gamma \vdash A \equiv B$$

What is conversion exactly?

Generated by hardwired basic equations on the language e.g.

- β-reduction: $(\lambda(x : A).\, M)\ N \equiv M\{x := N\}$
- pattern-matching reduction on constructors
- constant unfolding

Remember, type-checking should be decidable, so conversion as well.

$\rightsquigarrow$ in particular the kernel must implement conversion.

Liberté, égalité, prouvabilité

Why is conversion critical?

Liberté, égalité, prouvabilité

Why is conversion critical?

After all:

- Simple type theories like HOL do not have conversion*
- There are even "weak" dependent type theories without conversion

Liberté, égalité, prouvabilité

Why is conversion critical?

After all:

- Simple type theories like HOL do not have conversion*
- There are even "weak" dependent type theories without conversion

Conversion is both a blessing and a curse

Liberté, égalité, prouvabilité

Why is conversion critical?

After all:

- Simple type theories like HOL do not have conversion*
- There are even "weak" dependent type theories without conversion

Conversion is both a blessing and a curse

$\rightsquigarrow$ Why not take advantage of something that is automagic?

- Delegating from the user to the machine is the point of an assistant
- In HOL you must provide a proof (e.g. by rewriting tactics)
- Inefficient: you have to store it somehow

Liberté, égalité, prouvabilité

Why is conversion critical?

After all:

- Simple type theories like HOL do not have conversion*
- There are even “weak” dependent type theories without conversion

Conversion is both a blessing and a curse

$\rightsquigarrow$ Why not take advantage of something that is automagic?

- Delegating from the user to the machine is the point of an assistant
- In HOL you must provide a proof (e.g. by rewriting tactics)
- Inefficient: you have to store it somehow

$\rightsquigarrow$ Writing explicitly conversion derivations in CIC is not humanly possible.

Calculo Ergo Sum

Much better: we can take advantage of conversion.

Remember that CIC is a programming language? Let's put this to use.

Much better: we can take advantage of conversion.

Remember that CIC is a programming language? Let's put this to use.

Reflection

Replace logic by computation.

Idea: Assume some theory $\mathcal{T}$ that is decidable (or admits checkable proofs)

Calculo Ergo Sum

Much better: we can take advantage of conversion.

Remember that CIC is a programming language? Let's put this to use.

Reflection

Replace logic by computation.

Idea: Assume some theory $\mathcal{T}$ that is decidable (or admits checkable proofs)

- define a CIC AST `formula` representing $\mathcal{T}$-formulas

Calculo Ergo Sum

Much better: we can take advantage of conversion.

Remember that CIC is a programming language? Let's put this to use.

Reflection

Replace logic by computation.

Idea: Assume some theory $\mathcal{T}$ that is decidable (or admits checkable proofs)

- define a CIC AST `formula` representing $\mathcal{T}$-formulas
- write a CIC embedding $\texttt{eval} : \texttt{formula} \rightarrow \texttt{Prop}$

Much better: we can take advantage of conversion.

Remember that CIC is a programming language? Let's put this to use.

Reflection

Replace logic by computation.

Idea: Assume some theory $\mathcal{T}$ that is decidable (or admits checkable proofs)

- define a CIC AST `formula` representing $\mathcal{T}$-formulas
- write a CIC embedding $\texttt{eval} : \texttt{formula} \rightarrow \texttt{Prop}$
- write a CIC function $\texttt{check} : \texttt{formula} \rightarrow \mathbb{B}$

Calculo Ergo Sum

Much better: we can take advantage of conversion.

Remember that CIC is a programming language? Let's put this to use.

Reflection

Replace logic by computation.

Idea: Assume some theory $\mathcal{T}$ that is decidable (or admits checkable proofs)

- define a CIC AST `formula` representing $\mathcal{T}$-formulas
- write a CIC embedding $\mathtt{eval} : \mathtt{formula} \rightarrow \mathtt{Prop}$
- write a CIC function $\mathtt{check} : \mathtt{formula} \rightarrow \mathbb{B}$
- prove in CIC that $\Pi(\varphi : \mathtt{formula}).\, \mathtt{check}\ \varphi = \mathtt{true} \rightarrow \mathtt{eval}\ \varphi$

Much better: we can take advantage of conversion.

Remember that CIC is a programming language? Let's put this to use.

Reflection

Replace logic by computation.

Idea: Assume some theory $\mathcal{T}$ that is decidable (or admits checkable proofs)

- define a CIC AST `formula` representing $\mathcal{T}$-formulas
- write a CIC embedding $\mathtt{eval} : \mathtt{formula} \to \mathtt{Prop}$
- write a CIC function $\mathtt{check} : \mathtt{formula} \to \mathbb{B}$
- prove in CIC that $\Pi(\varphi : \mathtt{formula}).\, \mathtt{check}\ \varphi = \mathtt{true} \to \mathtt{eval}\ \varphi$

To prove $\Phi \in \mathcal{T}$ s.t. $\Phi := \mathtt{eval}\ \varphi$, it is thus enough to **compute** $\mathtt{check}\ \varphi$.

Sum Ergo Calculo

Reflection is a very powerful technique.

- Historically used for performance reasons
- Abstractly, a way to teach the kernel any theory (a logical JIT)

Sum Ergo Calculo

Reflection is a very powerful technique.

- Historically used for performance reasons
- Abstractly, a way to teach the kernel any theory (a logical JIT)

Many use cases: equational reasoning, arithmetic, SAT solving...

Sum Ergo Calculo

Reflection is a very powerful technique.

- Historically used for performance reasons
- Abstractly, a way to teach the kernel any theory (a logical JIT)

Many use cases: equational reasoning, arithmetic, SAT solving...

A related (but distinct) technique: small scale reflection.

- Similar idea of computing away trivial reasoning steps
- At the core of the SSReflect framework
- Famously used by Mathcomp and friends (e.g. Feit-Thompson proof)

Sum Ergo Calculo

Reflection is a very powerful technique.

- Historically used for performance reasons
- Abstractly, a way to teach the kernel any theory (a logical JIT)

Many use cases: equational reasoning, arithmetic, SAT solving...

A related (but distinct) technique: small scale reflection.

- Similar idea of computing away trivial reasoning steps
- At the core of the SSReflect framework
- Famously used by Mathcomp and friends (e.g. Feit-Thompson proof)

Morale

Computation matters!

Apostasie Trinitaire

The Coq kernel has not one but **three**† conversion algorithms.

Apostasie Trinitaire

The Coq kernel has not one but **three**† conversion algorithms.

$\rightsquigarrow$ The "reference" one

- Untyped, call-by-need reduction machine in OCaml
- Tailored for symbolic conversion, extremely sensitive

Apostasie Trinitaire

The Coq kernel has not one but **three**† conversion algorithms.

$\rightsquigarrow$ The “reference” one

- Untyped, call-by-need reduction machine in OCaml
- Tailored for symbolic conversion, extremely sensitive

$\rightsquigarrow$ The “heavy-duty” ones: VM and native

- Compilation to ZINC-like bytecode / machine code
- The VM is implemented in C, native reuses the OCaml runtime
- Tailored for brutal computation (typically, compute a boolean)

Apostasie Trinitaire

The Coq kernel has not one but **three**† conversion algorithms.

⇝ The "reference" one

- Untyped, call-by-need reduction machine in OCaml
- Tailored for symbolic conversion, extremely sensitive

⇝ The "heavy-duty" ones: VM and native

- Compilation to ZINC-like bytecode / machine code
- The VM is implemented in C, native reuses the OCaml runtime
- Tailored for brutal computation (typically, compute a boolean)

Different kind of trade-offs. What is the design space?

(For instance, Lean has an ad-hoc native-like process that only works on closed terms.)

Qui vit par le glaive...

We are not done with conversion yet!

Qui vit par le glaive...

We are not done with conversion yet!

Let $p, q : \{n : \mathbb{N} \mid \texttt{isEven}\ n\}$.

If $p.1 \equiv q.1$ then we do not have in general $p \equiv q$.

In pen-and-paper proofs one *never ever* cares about that.

Qui vit par le glaive...

We are not done with conversion yet!

Let $p, q : \{n : \mathbb{N} \mid \texttt{isEven}\ n\}$.

If $p.1 \equiv q.1$ then we do not have in general $p \equiv q$.

In pen-and-paper proofs one *never ever* cares about that.

The Dependent Hell

Proofs are programs, and thus relevant.

We would like **more** conversion!

Infère et Damnation

Not just a theoretical issue, this is a real PITA in practice.

Infère et Damnation

Not just a theoretical issue, this is a real PITA in practice.

Sibylline errors in innocuous scripts that require a PhD in type theory to understand.

```
Abstracting over the term "n" leads to a term
fun n0 : nat ⇒ exist (fun n1 : nat ⇒ isEven n1) n0 p = exist (fun n1 : nat ⇒ isEven n1) m q
which is ill−typed.
Reason is: Illegal application:
The term "exist" of type "forall (A : Type) (P : A → Prop) (x : A), P x → {x : A | P x}"
cannot be applied to the terms
"nat" : "Set" "fun n : nat ⇒ isEven n" : "nat → Prop" "n0" : "nat" "p" : "isEven n"
The 4th term has type "isEven n" which should be a subtype of "(fun n : nat ⇒ isEven n) n0".
(cannot unify "isEven n" and "isEven n0")
```

(This is just after `rewrite` e where $m, n : \mathbb{N}$ and $e : m = n$.)

Infère et Damnation

Not just a theoretical issue, this is a real PITA in practice.

Sibylline errors in innocuous scripts that require a PhD in type theory to understand.

```
Abstracting over the term "n" leads to a term
fun n0 : nat ⇒ exist (fun n1 : nat ⇒ isEven n1) n0 p = exist (fun n1 : nat ⇒ isEven n1) m q
which is ill-typed.
Reason is: Illegal application:
The term "exist" of type "forall (A : Type) (P : A → Prop) (x : A), P x → {x : A | P x}"
cannot be applied to the terms
"nat" : "Set" "fun n : nat ⇒ isEven n" : "nat → Prop" "n0" : "nat" "p" : "isEven n"
The 4th term has type "isEven n" which should be a subtype of "(fun n : nat ⇒ isEven n) n0".
(cannot unify "isEven n" and "isEven n0")
```

(This is just after `rewrite` e where $m, n : \mathbb{N}$ and $e : m = n$.)

A well-known problem that has plagued CIC for years

- Famous hazing for PhD students
- SSReflect even has a design pattern to work around the issue
- Outside of the kernel, not completely satisfactory

La Strictitude, c'est la Stricte Attitude

Recently solved by the introduction of a universe of strict propositions

- After all, proofs are not quite programs
- We don't care about proof contents: "all proofs are born equal."

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : A \qquad \Gamma \vdash A : \mathtt{SProp}}{\Gamma \vdash M \equiv N : A}$$

La Strictitude, c'est la Stricte Attitude

Recently solved by the introduction of a universe of strict propositions

- After all, proofs are not quite programs
- We don't care about proof contents: "all proofs are born equal."

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : A \qquad \Gamma \vdash A : \texttt{SProp}}{\Gamma \vdash M \equiv N : A}$$

The rules for `SProp` are tricky

- The feature was inspired by foundational work in HoTT
- Required non-trivial changes in the kernel
- Lean notoriously ~~doesn't give a shit~~ is practically-minded

Bien, mais pas top

`SProp` is a game changer

- An elegant solution to perennial issues
- Critical, but not enough

Bien, mais pas top

`SProp` is a game changer

- An elegant solution to perennial issues
- Critical, but not enough

There are many more limitations to conversion!

$$\mathtt{hd} : \Pi(n : \mathbb{N}).\, \mathtt{vec}\ A\ (1 + n) \rightarrow A \qquad\qquad v : \mathtt{vec}\ A\ (n + 1)$$

Bien, mais pas top

SProp is a game changer

- An elegant solution to perennial issues
- Critical, but not enough

There are many more limitations to conversion!

$$\mathtt{hd} : \Pi(n : \mathbb{N}).\, \mathtt{vec}\ A\ (1 + n) \to A \qquad v : \mathtt{vec}\ A\ (n + 1)$$

$\mathtt{hd}\ n\ v$ does not type-check because $1 + n \not\equiv n + 1$

This is much harder to solve.

Retour à l'ATER

Interestingly, these questions were fashionable in the '90s and 00's

- Extensionality in type theory (Hofmann)
- Observational type theory (Altenkirch-McBride)
- Coq Modulo Theory (Strub)

Retour à l'ATER

Interestingly, these questions were fashionable in the '90s and 00's

- Extensionality in type theory (Hofmann)
- Observational type theory (Altenkirch-McBride)
- Coq Modulo Theory (Strub)

... then nobody cared

- Systems were either not implemented or not used / not maintained

Retour à l'ATER

Interestingly, these questions were fashionable in the '90s and 00's

- Extensionality in type theory (Hofmann)
- Observational type theory (Altenkirch-McBride)
- Coq Modulo Theory (Strub)

... then nobody cared

- Systems were either not implemented or not used / not maintained

... but now it is making a comeback

- strict propositions
- rewrite rules
- extension types

Retour à l'ATER

Interestingly, these questions were fashionable in the '90s and 00's

- Extensionality in type theory (Hofmann)
- Observational type theory (Altenkirch-McBride)
- Coq Modulo Theory (Strub)

... then nobody cared

- Systems were either not implemented or not used / not maintained

... but now it is making a comeback

- strict propositions
- rewrite rules
- extension types

Will the cycle continue?

Universes

Ton Univers Impitoyable

In CIC, types are first-class citizens.

$\rightsquigarrow$ in particular, types have a universe type, traditionally called `Type`.

Ton Univers Impitoyable

In CIC, types are first-class citizens.

$\rightsquigarrow$ in particular, types have a universe type, traditionally called `Type`.

What is the type of `Type`?

In CIC, types are first-class citizens.

$\rightsquigarrow$ in particular, types have a universe type, traditionally called `Type`.

What is the type of `Type`?

Martin-Löf '71: $\mathtt{Type} : \mathtt{Type}$.

Ton Univers Impitoyable

In CIC, types are first-class citizens.

$\rightsquigarrow$ in particular, types have a universe type, traditionally called `Type`.

What is the type of `Type`?

Martin-Löf '71: `Type` : `Type`.

Girard '71 + ε: `Type` : `Type` is inconsistent.

Ton Univers Impitoyable

In CIC, types are first-class citizens.

$\leadsto$ in particular, types have a universe type, traditionally called $\mathtt{Type}$.

What is the type of $\mathtt{Type}$?

Martin-Löf '71: $\mathtt{Type} : \mathtt{Type}$.

Girard '71 $+ \varepsilon$: $\mathtt{Type} : \mathtt{Type}$ is inconsistent.

Standard solution: one has to stratify.

$$\mathtt{Type}_0 : \mathtt{Type}_1 : \mathtt{Type}_2 : \ldots : \mathtt{Type}_n : \mathtt{Type}_{n+1} : \ldots$$

We theoretical computer scientists love natural numbers!

$$(\mathtt{Type}_i)_{i \in \mathbb{N}}$$

- Simple
- Elegant
- Universal

We theoretical computer scientists love natural numbers!

$$(\mathtt{Type}_i)_{i \in \mathbb{N}}$$

- Simple
- Elegant
- Universal
- **... and completely anti-modular!**

Les Heures Sombres du BASIC

We theoretical computer scientists love natural numbers!

$$(\mathtt{Type}_i)_{i \in \mathbb{N}}$$

- Simple
- Elegant
- Universal
- **... and completely anti-modular!**

You have to pick all your levels upfront.

We theoretical computer scientists love natural numbers!

$$(\texttt{Type}_i)_{i\in\mathbb{N}}$$

- Simple
- Elegant
- Universal
- **… and completely anti-modular!**

You have to pick all your levels upfront.

If you have two universes $\texttt{Type}_n : \texttt{Type}_{n+1}$, you better not realize that you need some m s.t. $n < m < n+1$.

Les Heures Sombres du BASIC

We theoretical computer scientists love natural numbers!

$$(\mathtt{Type}_i)_{i \in \mathbb{N}}$$

- Simple
- Elegant
- Universal
- **... and completely anti-modular!**

You have to pick all your levels upfront.

If you have two universes $\mathtt{Type}_n : \mathtt{Type}_{n+1}$, you better not realize that you need some m s.t. $n < m < n + 1$.

This is why you should number your levels by increments of 100.

We theoretical computer scientists love natural numbers!

$$(\mathtt{Type}_i)_{i \in \mathbb{N}}$$

- Simple
- Elegant
- Universal
- **... and completely anti-modular!**

You have to pick all your levels upfront.

If you have two universes $\mathtt{Type}_n : \mathtt{Type}_{n+1}$, you better not realize that you need some m s.t. $n < m < n + 1$.

~~This is why you should number your levels by increments of 100.~~

Ça varie

Floating universes (antediluvian trick)

Just introduce variables

Ça varie

Floating universes (antediluvian trick)

Just introduce variables

The language is simple:

- Variable universe levels i, j...
- Constraints $i < j$ and $i \leq j$ which must form a DAG

Ça varie

Floating universes (antediluvian trick)

Just introduce variables

The language is simple:

- Variable universe levels $i, j\ldots$
- Constraints $i < j$ and $i \leq j$ which must form a DAG

Minor tweaks to the kernel

- Generate fresh levels (outside of the kernel)
- Accumulate constraints (outside of the kernel)
- Send the graph to the kernel for acyclicity checking

Ça varie

Floating universes (antediluvian trick)

Just introduce variables

The language is simple:

- Variable universe levels $i, j\ldots$
- Constraints $i < j$ and $i \leq j$ which must form a DAG

Minor tweaks to the kernel

- Generate fresh levels (outside of the kernel)
- Accumulate constraints (outside of the kernel)
- Send the graph to the kernel for acyclicity checking

Voilá, you never have to care about universes again

Ça varie

Floating universes (antediluvian trick)

Just introduce variables

The language is simple:

- Variable universe levels $i, j...$
- Constraints $i < j$ and $i \leq j$ which must form a DAG

Minor tweaks to the kernel

- Generate fresh levels (outside of the kernel)
- Accumulate constraints (outside of the kernel)
- Send the graph to the kernel for acyclicity checking

Voilá, you never have to care about universes again

(By the way Agda and Lean have a different approach.)

L'uni vert de rage

This is a very wasteful technique

$\rightsquigarrow$ In 99% of the actual developments:

L'uni vert de rage

This is a very wasteful technique

$\rightsquigarrow$ In 99% of the actual developments:

You have to generate a bazillion universe levels

- Most of them are transient
- They are generated by tactics and unified away immediately
- You only have to send a gazillion levels to the kernel

L'uni vert de rage

This is a very wasteful technique

$\rightsquigarrow$ In 99% of the actual developments:

You have to generate a bazillion universe levels

- Most of them are transient
- They are generated by tactics and unified away immediately
- You only have to send a gazillion levels to the kernel

The gazillion-sized graph is a fiction. Collapsing $\leq$ constraints gives:

$$i \qquad < \qquad j \qquad < \qquad k$$

L'uni vert de rage

This is a very wasteful technique

$\rightsquigarrow$ In 99% of the actual developments:

You have to generate a bazillion universe levels

- Most of them are transient
- They are generated by tactics and unified away immediately
- You only have to send a gazillion levels to the kernel

The gazillion-sized graph is a fiction. Collapsing $\leq$ constraints gives:

$$i \qquad < \qquad j \qquad < \qquad k$$

Three levels ought to be enough for anybody!

Trop c'est trop

This is an actually very annoying problem

Users have been complaining about related performance issues for ages.

Even when you are not aware, you pay for this.

Trop c'est trop

This is an actually very annoying problem

Users have been complaining about related performance issues for ages.

Even when you are not aware, you pay for this.

Universe generation is also very much inscrutable.

It is effectful and incompatible with desirable features.

Good luck debugging a stray constraint.

Trop c'est trop

This is an actually very annoying problem

Users have been complaining about related performance issues for ages.

Even when you are not aware, you pay for this.

Universe generation is also very much inscrutable.

It is effectful and incompatible with desirable features.

Good luck debugging a stray constraint.

Horror Story

The MetaCoq and QuickChick are (were?) not loadable together.

Une auberge dont on n'est pas sorti

Worse!

Une auberge dont on n'est pas sorti

Worse!

Serious Question: What is the type of $\texttt{Type}_i$?

Une auberge dont on n'est pas sorti

Worse!

Serious Question:	What is the type of $\texttt{Type}_i$?
Timid Answer:	$\texttt{Type}_j$ for some $j > i$?

Une auberge dont on n'est pas sorti

Worse!

Serious Question: What is the type of $\texttt{Type}_i$?
Timid Answer: $\texttt{Type}_j$ for some $j > i$?

No! You shouldn't be able to generate fresh levels from within the kernel.

Une auberge dont on n'est pas sorti

Worse!

Serious Question:	What is the type of $\mathtt{Type}_i$?
Timid Answer:	$\mathtt{Type}_j$ for some $j > i$?

No! You shouldn't be able to generate fresh levels from within the kernel.

I lied (again)

We still need algebraic universe expressions in types.

- In Coq, types are *actually* not terms!
- Some kind of adjunction between types and terms

$$\exists j > i.\, \mathtt{Type}_j \qquad \sim \qquad \mathtt{Type}_{i+1}$$

Une auberge dont on n'est pas sorti

Worse!

Serious Question:	What is the type of $\texttt{Type}_i$?
Timid Answer:	$\texttt{Type}_j$ for some $j > i$?

No! You shouldn't be able to generate fresh levels from within the kernel.

I lied (again)

We still need algebraic universe expressions in types.

- In Coq, types are *actually* not terms!
- Some kind of adjunction between types and terms

$$\exists j > i.\, \texttt{Type}_j \qquad \sim \qquad \texttt{Type}_{i+1}$$

The Andouillette Principle

I am not sure I have seen this really publicized anywhere.

Un nouvel espoir

This universe business is currently a hot topic

- Yet another universe checking algorithm
- That handles algebraic universes natively

Un nouvel espoir

This universe business is currently a hot topic

- Yet another universe checking algorithm
- That handles algebraic universes natively

Resonates with the multiverse project

- More complex sorts, expanding the logic
- All of this is tied together inextricably

Un nouvel espoir

This universe business is currently a hot topic

- Yet another universe checking algorithm
- That handles algebraic universes natively

Resonates with the multiverse project

- More complex sorts, expanding the logic
- All of this is tied together inextricably

Yet another revival of dormant questions

Guard

Totalitarisme logique

In CIC, all functions must terminate.

- Otherwise the theory becomes inconsistent.
- Arbitrary fixpoints allow $n : \mathbb{N}$ s.t. $n := \mathsf{S}\ n$

Totalitarisme logique

In CIC, all functions must terminate.

- Otherwise the theory becomes inconsistent.
- Arbitrary fixpoints allow $n : \mathbb{N}$ s.t. $n := \mathsf{S}\ n$

In paper presentations, for simplicity one uses recursors.

$$\begin{array}{l} \mathtt{rec}_{\mathbb{N}} : P\ \mathsf{O} \rightarrow (\Pi(n : \mathbb{N}).\ P\ n \rightarrow P\ (\mathsf{S}\ n)) \rightarrow \Pi(n : \mathbb{N}).\ P\ n \\ \qquad \begin{array}{lcl} \mathtt{rec}_{\mathbb{N}}\ p_O\ p_S\ \mathsf{O} & \equiv & p_O \\ \mathtt{rec}_{\mathbb{N}}\ p_O\ p_S\ (\mathsf{S}\ n) & \equiv & p_S\ n\ (\mathtt{rec}_{\mathbb{N}}\ n\ p_O\ p_S) \end{array} \end{array}$$

Totalitarisme logique

In CIC, all functions must terminate.

- Otherwise the theory becomes inconsistent.
- Arbitrary fixpoints allow $n : \mathbb{N}$ s.t. $n := \mathsf{S}\ n$

In paper presentations, for simplicity one uses recursors.

$$\begin{array}{l} \mathtt{rec}_{\mathbb{N}} : P\ \mathsf{O} \to (\Pi(n:\mathbb{N}).\, P\ n \to P\ (\mathsf{S}\ n)) \to \Pi(n:\mathbb{N}).\, P\ n \\ \qquad \begin{array}{lcl} \mathtt{rec}_{\mathbb{N}}\ p_O\ p_S\ \mathsf{O} & \equiv & p_O \\ \mathtt{rec}_{\mathbb{N}}\ p_O\ p_S\ (\mathsf{S}\ n) & \equiv & p_S\ n\ (\mathtt{rec}_{\mathbb{N}}\ n\ p_O\ p_S) \end{array} \end{array}$$

The Category Terrorist

"$\mathtt{rec}_{\mathbb{N}}$ is universal, because this is the universal property of $\mathbb{N}$."

Mon diagramme, il commute

The Category Terrorist

"$\mathtt{rec}_{\mathbb{N}}$ is universal, because this is the universal property of $\mathbb{N}$."

Mon diagramme, il commute

The Category Terrorist

"$\texttt{rec}_{\mathbb{N}}$ is universal, because this is the universal property of $\mathbb{N}$."

This is only true extensionally!

- This only holds for propositional equality
- Further assuming various extensionality principles

Mon diagramme, il commute

The Category Terrorist

"$\texttt{rec}_{\mathbb{N}}$ is universal, because this is the universal property of $\mathbb{N}$."

This is only true extensionally!

- This only holds for propositional equality
- Further assuming various extensionality principles

Computation matters

Would you conflate a $O(2^n)$ algorithm with $O(1)$ one?

- Intensional behaviour is critical for programming
- Recursors are very bad in call-by-value
- It is not even clear what universality means for conversion
- Whatever this means, recursors are not universal for it

Une idée fixe

Good news: recursors are not fundamental in Coq.

Une idée fixe

Good news: recursors are not fundamental in Coq.

Instead, Coq relies on fixpoints + pattern-matching.

$$\begin{array}{lcl} \mathtt{rec}_{\mathbb{N}}\ p_O\ p_S & := & \mathtt{fix}\ F\,(n : \mathbb{N}) := \mathtt{match}\ n\ \mathtt{with} \\ & & \qquad\qquad\qquad\quad \mid \mathsf{O} \Rightarrow p_O \\ & & \qquad\qquad\qquad\quad \mid \mathsf{S}\ m \Rightarrow p_S\ m\ (F\ m) \end{array}$$

Une idée fixe

Good news: recursors are not fundamental in Coq.

Instead, Coq relies on fixpoints + pattern-matching.

$$\begin{array}{lll} \mathtt{rec}_{\mathbb{N}}\ p_O\ p_S & := & \mathtt{fix}\ F\,(n : \mathbb{N}) := \mathtt{match}\ n\ \mathtt{with} \\ & & \qquad\qquad\qquad\ \ \mid \mathsf{O} \Rightarrow p_O \\ & & \qquad\qquad\qquad\ \ \mid \mathsf{S}\ m \Rightarrow p_S\ m\ (F\ m) \end{array}$$

This is a historical design choice motivated by extraction

- Similar to OCaml
- The extracted terms look like what the user wrote
- Critical for efficiency in call-by-value

Une idée fixe

Good news: recursors are not fundamental in Coq.

Instead, Coq relies on fixpoints + pattern-matching.

$$\begin{array}{lcl} \mathtt{rec}_{\mathbb{N}}\ p_O\ p_S & := & \mathtt{fix}\ F\,(n : \mathbb{N}) := \mathtt{match}\ n\ \mathtt{with} \\ & & \qquad\qquad\qquad\quad \mid \mathsf{O} \Rightarrow p_O \\ & & \qquad\qquad\qquad\quad \mid \mathsf{S}\ m \Rightarrow p_S\ m\ (F\ m) \end{array}$$

This is a historical design choice motivated by extraction

- Similar to OCaml
- The extracted terms look like what the user wrote
- Critical for efficiency in call-by-value

One can write fixpoints that are not intensionally recursor-encodable.

$$\mathtt{even} : \mathbb{N} \to \mathbb{B}$$

$$\begin{array}{lcl} \mathtt{even}\ \mathsf{O} & := & \mathsf{true} \\ \mathtt{even}\ (\mathsf{S}\ \mathsf{O}) & := & \mathsf{false} \\ \mathtt{even}\ (\mathsf{S}\ (\mathsf{S}\ n)) & := & \mathtt{even}\ n \end{array}$$

Si tu ne viens pas à Lagardère

Bad news: recursors are not fundamental in Coq.

Si tu ne viens pas à Lagardère

Bad news: recursors are not fundamental in Coq.

Consider the following:

- Coq relies on fixpoints + pattern-matching.
- Arbitrary fixpoints are inconsistent.

Si tu ne viens pas à Lagardère

Bad news: recursors are not fundamental in Coq.

Consider the following:

- Coq relies on fixpoints + pattern-matching.
- Arbitrary fixpoints are inconsistent.

Hence there must be some mechanism to restrict to *good* fixpoints

Si tu ne viens pas à Lagardère

Bad news: recursors are not fundamental in Coq.

Consider the following:

- Coq relies on fixpoints + pattern-matching.
- Arbitrary fixpoints are inconsistent.

Hence there must be some mechanism to restrict to *good* fixpoints

La garde!

La programmation totale, c'est totalement con

What does the guard enforce?

Minimal service:

- The theory must be consistent
- Hence functions ought to be total.

La programmation totale, c'est totalement con

What does the guard enforce?

Minimal service:

- The theory must be consistent
- Hence functions ought to be total.

Once again, the MetaCoq people worked this out a bit.

- Various closure conditions
- Some surprisingly non-necessary properties

La programmation totale, c'est totalement con

What does the guard enforce?

Minimal service:

- The theory must be consistent
- Hence functions ought to be total.

Once again, the MetaCoq people worked this out a bit.

- Various closure conditions
- Some surprisingly non-necessary properties

The more expressive the guard, the better.

(Right?)

Terminons-en

The guard condition is probably the least understood kernel component.

- Specification not quite clear, stay tuned
- Organic implementation — *it would be nice if this worked...*
- Decades of tweaks and RFC from users
- **... and obviously critical for consistency**

Terminons-en

The guard condition is probably the least understood kernel component.

- Specification not quite clear, stay tuned
- Organic implementation — *it would be nice if this worked...*
- Decades of tweaks and RFC from users
- **... and obviously critical for consistency**

I want to give you a foretaste of kern-hell.

La garde meurt et ne retourne jamais

Ingenuous question

What does total mean?

La garde meurt et ne retourne jamais

Ingenuous question

What does total mean?

In CIC, the equational theory is call-by-name.

$\rightsquigarrow$ In particular, we only care about weak-head reduction.

La garde meurt et ne retourne jamais

Ingenuous question

What does total mean?

In CIC, the equational theory is call-by-name.

$\leadsto$ In particular, we only care about weak-head reduction.

This used to be accepted

$$\texttt{fix loop}\ (i : \texttt{unit}) := \texttt{let}\ \_ := \texttt{loop}\ ()\ \texttt{in}\ ()$$

- Perfectly fine in call-by-name
- Not inconsistent, this is just a constant function
- Not quite so in call-by-value, e.g. through extraction

La garde meurt et ne retourne jamais

Ingenuous question

What does total mean?

In CIC, the equational theory is call-by-name.

$\rightsquigarrow$ In particular, we only care about weak-head reduction.

This used to be accepted

$$\texttt{fix loop}\ (i : \texttt{unit}) := \texttt{let}\ \_ := \texttt{loop}\ ()\ \texttt{in}\ ()$$

- Perfectly fine in call-by-name
- Not inconsistent, this is just a constant function
- Not quite so in call-by-value, e.g. through extraction

As of Coq 8.19, not accepted, but still morally OK in the abstract.

Ouate le Foulque

Logically, what is the worse you could get from a defective guard?

Ouate le Foulque

Logically, what is the worse you could get from a defective guard?

"Morally, you could be inconsistent. There should not be anything in between. Apart from more functions, that is." — Sweet Summer Child.

Ouate le Foulque

Logically, what is the worse you could get from a defective guard?

"Morally, you could be inconsistent. There should not be anything in between. Apart from more functions, that is." — Sweet Summer Child.

The guard condition used to negate propositional extensionality.

$$\mathtt{PropExt} := \Pi(P\,Q : \mathtt{Prop}).\,(P \leftrightarrow Q) \rightarrow P = Q$$

$\rightsquigarrow$ this is inconsistent with both HoTT and the $\mathbf{Set}$ model

Ouate le Foulque

Logically, what is the worse you could get from a defective guard?

"Morally, you could be inconsistent. There should not be anything in between. Apart from more functions, that is." — Sweet Summer Child.

The guard condition used to negate propositional extensionality.

$$\mathtt{PropExt} := \Pi(P\,Q : \mathtt{Prop}).\,(P \leftrightarrow Q) \rightarrow P = Q$$

$\leadsto$ this is inconsistent with both HoTT and the $\mathbf{Set}$ model

In the name of Gödel, what does this have to do with termination?

Ouate le Foulque

Logically, what is the worse you could get from a defective guard?

"Morally, you could be inconsistent. There should not be anything in between. Apart from more functions, that is." — Sweet Summer Child.

The guard condition used to negate propositional extensionality.

$$\texttt{PropExt} := \Pi(P\,Q : \texttt{Prop}).\,(P \leftrightarrow Q) \rightarrow P = Q$$

$\rightsquigarrow$ this is inconsistent with both HoTT and the **Set** model

In the name of Gödel, what does this have to do with termination?

(Mumble something about size issues.)

Le feu ça brûle et l'eau ça mouille

Consistency is not enough!

Le feu ça brûle et l'eau ça mouille

Consistency is not enough!

We want the guard to be as neutral as possible w.r.t. model validity...

- HoTT
- **Set**
- Something else?

Le feu ça brûle et l'eau ça mouille

Consistency is not enough!

We want the guard to be as neutral as possible w.r.t. model validity…

- HoTT
- **Set**
- Something else?

… but we also want it to be as expressive as possible.

- Some people out there make a living of this

Le feu ça brûle et l'eau ça mouille

Consistency is not enough!

We want the guard to be as neutral as possible w.r.t. model validity...

- HoTT
- **Set**
- Something else?

... but we also want it to be as expressive as possible.

- Some people out there make a living of this

This is not a formal specification!

Courage, fuyons

The only idealized model we understand is recursor-based†.

Courage, fuyons

The only idealized model we understand is recursor-based[†].

([†] This is not even completely true. If you hear the word *nested*, run.)

Courage, fuyons

The only idealized model we understand is recursor-based†.

(†This is not even completely true. If you hear the word *nested*, run.)

$\rightsquigarrow$ We should be able to justify the guard by compilation to recursors.

- This could even be done outside of the kernel
- This is actually used by e.g. Equations

Courage, fuyons

The only idealized model we understand is recursor-based†.

(†This is not even completely true. If you hear the word *nested*, run.)

$\leadsto$ We should be able to justify the guard by compilation to recursors.

- This could even be done outside of the kernel
- This is actually used by e.g. Equations

But this is doomed to fail: there are fixpoints not recursor-encodable.

$\leadsto$ At least not with an intensional notion of encodable.

- Some Coq-definable fixpoints conflict with recursor models
- Effects are pretty much incompatible with some guard assumptions
- What is a good notion of encodable?

Courage, fuyons

The only idealized model we understand is recursor-based†.

(†This is not even completely true. If you hear the word *nested*, run.)

$\rightsquigarrow$ We should be able to justify the guard by compilation to recursors.

- This could even be done outside of the kernel
- This is actually used by e.g. Equations

But this is doomed to fail: there are fixpoints not recursor-encodable.

$\rightsquigarrow$ At least not with an intensional notion of encodable.

- Some Coq-definable fixpoints conflict with recursor models
- Effects are pretty much incompatible with some guard assumptions
- What is a good notion of encodable?

Who shall guard the guard?

Conclusion

Bad Tripes?

The confidence of the theoretician crashes against the wall of reality

- The real has much more asperities
- Even in an idealized kernel lurk unknown monsters
- Diachronical and interindividual dialectics are pervasive

Bad Tripes?

The confidence of the theoretician crashes against the wall of reality

- The real has much more asperities
- Even in an idealized kernel lurk unknown monsters
- Diachronical and interindividual dialectics are pervasive

So is the essence of andouillette.

- Proof assistants are still an invaluable tool
- The andouillette principle should not be feared
- Rather, this is a never ending material for a thriving research

Bad Tripes?

The confidence of the theoretician crashes against the wall of reality

- The real has much more asperities
- Even in an idealized kernel lurk unknown monsters
- Diachronical and interindividual dialectics are pervasive

So is the essence of andouillette.

- Proof assistants are still an invaluable tool
- The andouillette principle should not be feared
- Rather, this is a never ending material for a thriving research

Do not be afraid and join us

Scribitur ad narrandum, non ad probandum.

L'absurde ne délivre pas, il lie. Il n'autorise pas tous les actes. Tout est permis ne signifie pas que rien n'est défendu.

Albert Camus.

Thank you for your attention.