

Is it time for a new proof assistant?

Jon Sterling

Cambridge Computer Laboratory

September 25, 2025

HoTT  Seminar Talks

Where we are.

There has never been a better time for proof assistants based on dependent type theory.

Where we are.

There has never been a better time for proof assistants based on dependent type theory.

- + **Lean** is a best-in-class proof assistant for classical mathematics, and comes with **Mathlib**. Huge investment in tooling and quality-of-life by **Lean FRO**.

Where we are.

There has never been a better time for proof assistants based on dependent type theory.

- + **Lean** is a best-in-class proof assistant for classical mathematics, and comes with **Mathlib**. Huge investment in tooling and quality-of-life by **Lean FRO**.
- + **Agda** is a proving ground for innovative type theories (e.g. cubical type theory), and the home of several strong libraries of mathematics in HoTT/UF: **agda-unimath**, **TypeTopology**, **1Lab**, and **cubical**.

Where we are.

There has never been a better time for proof assistants based on dependent type theory.

- + **Lean** is a best-in-class proof assistant for classical mathematics, and comes with **Mathlib**. Huge investment in tooling and quality-of-life by **Lean FRO**.
- + **Agda** is a proving ground for innovative type theories (*e.g.* cubical type theory), and the home of several strong libraries of mathematics in HoTT/UF: **agda-unimath**, **TypeTopology**, **1Lab**, and **cubical**.
- + **Rocq** is admittedly a little long in the tooth, but continues to be a strong player in program verification (*e.g.* Iris). Notable HoTT libraries: **UniMath**, **Coq-HoTT**, *etc.*

Where we are.

There has never been a better time for proof assistants based on dependent type theory.

- + **Lean** is a best-in-class proof assistant for classical mathematics, and comes with **Mathlib**. Huge investment in tooling and quality-of-life by **Lean FRO**.
- + **Agda** is a proving ground for innovative type theories (e.g. cubical type theory), and the home of several strong libraries of mathematics in HoTT/UF: **agda-unimath**, **TypeTopology**, **1Lab**, and **cubical**.
- + **Rocq** is admittedly a little long in the tooth, but continues to be a strong player in program verification (e.g. Iris). Notable HoTT libraries: **UniMath**, **Coq-HoTT**, *etc.*
- + Some experimental systems, notably **Idris 2** for **Quantitative Type Theory**, **Narya** for **Higher Observational Type Theory**, and **Istari** for **Impredicative Computational Type Theory**.

What about HoTT/UF?

Most of the current energy is invested in Agda and Rocq, split in two communities:

1. **Cubical**: two mathematical libraries built in the *Cubical Agda* dialect, namely the **1Lab** and the **cubical** library.
2. “**Orthodox**”: some prefer to work in the dialect of the Book: **UniMath**, **agda-unimath**, **TypeTopology**, **Coq-HoTT**.

After the initial flurry in the 2010s, we never managed to retain interest from mainstream mathematicians, who have overwhelmingly flocked to Lean.

The Eighteenth Brumaire of...formal mathematics

Looking back at the hype cycles for both HoTT and Lean, one might conclude that the main way to attract mathematicians is through **affinity-based publicity campaigns**.

The Eighteenth Brumaire of... formal mathematics

Looking back at the hype cycles for both HoTT and Lean, one might conclude that the main way to attract mathematicians is through **affinity-based publicity campaigns**.

- + **Vladimir Voevodsky** stirred up a hurricane of interest in formalisation, in the ensuing hype, some mathematicians got the message that HoTT/UF was *the only reliable way* to formalise modern mathematics. This was not true.

The Eighteenth Brumaire of... formal mathematics

Looking back at the hype cycles for both HoTT and Lean, one might conclude that the main way to attract mathematicians is through **affinity-based publicity campaigns**.

- + **Vladimir Voevodsky** stirred up a hurricane of interest in formalisation, in the ensuing hype, some mathematicians got the message that HoTT/UF was *the only reliable way* to formalise modern mathematics. This was not true.
- + In time for the backlash, **Kevin Buzzard** stirred up new interest in formalisation via Lean—at times by attributing to Lean what was already present in Rocq, and by making inaccurate statements about a supposed tension between UF and classical mathematics.

The Eighteenth Brumaire of... formal mathematics

Looking back at the hype cycles for both HoTT and Lean, one might conclude that the main way to attract mathematicians is through **affinity-based publicity campaigns**.

- + **Vladimir Voevodsky** stirred up a hurricane of interest in formalisation, in the ensuing hype, some mathematicians got the message that HoTT/UF was *the only reliable way* to formalise modern mathematics. This was not true.
- + In time for the backlash, **Kevin Buzzard** stirred up new interest in formalisation via Lean—at times by attributing to Lean what was already present in Rocq, and by making inaccurate statements about a supposed tension between UF and classical mathematics.

Today, ill-informed **AI Hype** is all the rage among the “Fields Medal set”. Must we bend the knee?

Taking our lumps: why Lean is great

Hype is not the only reason mathematicians have flocked to Lean.

Taking our lumps: why Lean is great

Hype is not the only reason mathematicians have flocked to Lean.

1. You can get Lean installed and running in **less than five seconds** from VS Code, on all platforms. I am not kidding.

Taking our lumps: why Lean is great

Hype is not the only reason mathematicians have flocked to Lean.

1. You can get Lean installed and running in **less than five seconds** from VS Code, on all platforms. I am not kidding.
2. Lean has **fantastic editor support** out of the box, and you don't need to use Emacs.

Taking our lumps: why Lean is great

Hype is not the only reason mathematicians have flocked to Lean.

1. You can get Lean installed and running in **less than five seconds** from VS Code, on all platforms. I am not kidding.
2. Lean has **fantastic editor support** out of the box, and you don't need to use Emacs.
3. Lean has **fast and somewhat reliable type classes** that let you get to work doing normal mathematics right away without spending hours examining trade-offs between differently unworkable library architectures.

Taking our lumps: why Lean is great

Hype is not the only reason mathematicians have flocked to Lean.

1. You can get Lean installed and running in **less than five seconds** from VS Code, on all platforms. I am not kidding.
2. Lean has **fantastic editor support** out of the box, and you don't need to use Emacs.
3. Lean has **fast and somewhat reliable type classes** that let you get to work doing normal mathematics right away without spending hours examining trade-offs between differently unworkable library architectures.
4. Lean has the excellent **Mathlib** library, which focuses unabashedly on the canon of standard mathematics.

Taking our lumps: why Lean is great

Hype is not the only reason mathematicians have flocked to Lean.

1. You can get Lean installed and running in **less than five seconds** from VS Code, on all platforms. I am not kidding.
2. Lean has **fantastic editor support** out of the box, and you don't need to use Emacs.
3. Lean has **fast and somewhat reliable type classes** that let you get to work doing normal mathematics right away without spending hours examining trade-offs between differently unworkable library architectures.
4. Lean has the excellent **Mathlib** library, which focuses unabashedly on the canon of standard mathematics.
5. *If Lean proves \perp , that's a **bug that will be fixed**. When Agda proves \perp , **that is a feature** (which we can't remove, because some weird Swedish code is using it!).*

What have we been doing in the meanwhile?

There was some truth to it when Kevin Buzzard said (paraphrased) that a great deal of energy in the HoTT/UF community has been focused on solving problems of **logic** rather than problems of **mathematics**. (Example: my thesis.)

(But it's not exclusively so! Many of us *have* been working on mathematics! And both!)

What have we been doing in the meanwhile?

There was some truth to it when Kevin Buzzard said (paraphrased) that a great deal of energy in the HoTT/UF community has been focused on solving problems of **logic** rather than problems of **mathematics**. (Example: my thesis.)

(But it's not exclusively so! Many of us *have* been working on mathematics! And both!)

I would instead say that logical and mathematical problems are part of the same body, but that **mathematics is the dog** and **logic is the tail**. Once in a while we should ask ourselves, who is wagging whom.

10 years of cubical type theory

As a community, we solved the problem of the **computational interpretation of univalent foundations**. But...

10 years of cubical type theory

As a community, we solved the problem of the **computational interpretation of univalent foundations**. But...

- + A computational interpretation is not necessary for *doing mathematics*. In fact, we usually want to *stop* computation from happening beyond a certain point; and cubical computation leaves behind undesirable junk, like dead coercions.

10 years of cubical type theory

As a community, we solved the problem of the **computational interpretation of univalent foundations**. But...

- + A computational interpretation is not necessary for *doing mathematics*. In fact, we usually want to *stop* computation from happening beyond a certain point; and cubical computation leaves behind undesirable junk, like dead coercions.
- + Cubical Agda is probably[?] not sound for standard mathematics because of the regrettable use of De Morgan cubes.

10 years of cubical type theory

As a community, we solved the problem of the **computational interpretation of univalent foundations**. But...

- + A computational interpretation is not necessary for *doing mathematics*. In fact, we usually want to *stop* computation from happening beyond a certain point; and cubical computation leaves behind undesirable junk, like dead coercions.
- + Cubical Agda is probably[?] not sound for standard mathematics because of the regrettable use of De Morgan cubes.
- + Even if we switch to a version of cubical type theory that is sound for standard mathematics (as in **redtt**, **cooltt**, etc.), cubical type theory has usability problems and is probably[?] incompatible with reliable implicit resolution.

10 years of cubical type theory

As a community, we solved the problem of the **computational interpretation of univalent foundations**. But...

- + A computational interpretation is not necessary for *doing mathematics*. In fact, we usually want to *stop* computation from happening beyond a certain point; and cubical computation leaves behind undesirable junk, like dead coercions.
- + Cubical Agda is probably? not sound for standard mathematics because of the regrettable use of De Morgan cubes.
- + Even if we switch to a version of cubical type theory that is sound for standard mathematics (as in **redtt**, **cooltt**, etc.), cubical type theory has usability problems and is probably? incompatible with reliable implicit resolution.

Cubical **models** more important than the cubical **type theory**?

15 years of UniMath: the jewel of HoTT/UF

A precious legacy.

- + **2003**: Vladimir Voevodsky calls for a computerised “Bourbaki project”—speaking of the generational transition from **text** to **hypertext** to **computer verified mathematics**.

15 years of UniMath: the jewel of HoTT/UF

A precious legacy.

- + **2003:** Vladimir Voevodsky calls for a computerised “Bourbaki project”—speaking of the generational transition from **text** to **hypertext** to **computer verified mathematics**.
- + **2014:** The **UniMath** project is born: a unified library of **general mathematics** from the point of view of univalent foundations in Rocq.

15 years of UniMath: the jewel of HoTT/UF

A precious legacy.

- + **2003**: Vladimir Voevodsky calls for a computerised “Bourbaki project”—speaking of the generational transition from **text** to **hypertext** to **computer verified mathematics**.
- + **2014**: The **UniMath** project is born: a unified library of **general mathematics** from the point of view of univalent foundations in Rocq.
- + **2021**: Egbert Rijke founds the **agda-unimath** library on similar design principles. A big success IMO, and an on-ramp for many students.

UniMath has thrived in our small community, but if there is a computerised Bourbaki project today, it is really Lean's **MathLib**.

UniMath has thrived in our small community, but if there is a computerised Bourbaki project today, it is really Lean's **MathLib**.

Why hasn't **UniMath** taken off outside our community? Maybe in part because the **style of formalisation is very low-level** and involves tons of plumbing.

UniMath has thrived in our small community, but if there is a computerised Bourbaki project today, it is really Lean's **MathLib**.

Why hasn't **UniMath** taken off outside our community? Maybe in part because the **style of formalisation is very low-level** and involves tons of plumbing.

Voevodsky chose to expose the plumbing in part because of a perception that the high-level features of Rocq and Agda are of...questionable semantic validity and portability.

UniMath has thrived in our small community, but if there is a computerised Bourbaki project today, it is really Lean's **MathLib**.

Why hasn't **UniMath** taken off outside our community? Maybe in part because the **style of formalisation is very low-level** and involves tons of plumbing.

Voevodsky chose to expose the plumbing in part because of a perception that the high-level features of Rocq and Agda are of...questionable semantic validity and portability.

Thesis: *A full solution to the underlying problems noted by Voevodsky must factor through a **return to foundational orthodoxy**, but we need not sacrifice usability.*

A thought experiment

I still believe in HoTT/UF and computer-assisted proof.

A thought experiment

I still believe in HoTT/UF and computer-assisted proof. But I want:

- + ...to stop wearing the hair shirt.

A thought experiment

I still believe in HoTT/UF and computer-assisted proof. But I want:

- + ...to stop wearing the hair shirt.

⇒ I need reliable and flexible tools for algebraic hierarchies. I need reliable control over definitional unfolding and the display of abstractions.

A thought experiment

I still believe in HoTT/UF and computer-assisted proof. But I want:

- + ...to stop wearing the hair shirt.
 - ⇒ I need reliable and flexible tools for algebraic hierarchies. I need reliable control over definitional unfolding and the display of abstractions.
- + ...to use it to teach undergraduates.

A thought experiment

I still believe in HoTT/UF and computer-assisted proof. But I want:

- + ...to stop wearing the hair shirt.
 - ⇒ I need reliable and flexible tools for algebraic hierarchies. I need reliable control over definitional unfolding and the display of abstractions.
- + ...to use it to teach undergraduates.
 - ⇒ I need easy installation, great editor support, and to minimise “type theoretic” artefacts that distract from the mathematics; goal-driven development; documentary “history” of deductions.

A thought experiment

I still believe in HoTT/UF and computer-assisted proof. But I want:

- + ...to stop wearing the hair shirt.
 - ⇒ I need reliable and flexible tools for algebraic hierarchies. I need reliable control over definitional unfolding and the display of abstractions.
- + ...to use it to teach undergraduates.
 - ⇒ I need easy installation, great editor support, and to minimise “type theoretic” artefacts that distract from the mathematics; goal-driven development; documentary “history” of deductions.
- + ...to use it to communicate with “normal” mathematicians.

A thought experiment

I still believe in HoTT/UF and computer-assisted proof. But I want:

- + ...to stop wearing the hair shirt.
 - ⇒ I need reliable and flexible tools for algebraic hierarchies. I need reliable control over definitional unfolding and the display of abstractions.
- + ...to use it to teach undergraduates.
 - ⇒ I need easy installation, great editor support, and to minimise “type theoretic” artefacts that distract from the mathematics; goal-driven development; documentary “history” of deductions.
- + ...to use it to communicate with “normal” mathematicians.
 - ⇒ I need the thing to be *clearly* sound for standard mathematics.

A thought experiment

I still believe in HoTT/UF and computer-assisted proof. But I want:

- + ...to stop wearing the hair shirt.
 - ⇒ I need reliable and flexible tools for algebraic hierarchies. I need reliable control over definitional unfolding and the display of abstractions.
- + ...to use it to teach undergraduates.
 - ⇒ I need easy installation, great editor support, and to minimise “type theoretic” artefacts that distract from the mathematics; goal-driven development; documentary “history” of deductions.
- + ...to use it to communicate with “normal” mathematicians.
 - ⇒ I need the thing to be *clearly* sound for standard mathematics.

All of this suggests to me that the best path forward **for my needs** is a **supercharged** implementation of Book HoTT. **Let's innovate on the tool, not on the core theory which is already great.**

Why not **H**igher **O**bservational **T**ype **T**heory?



Why not **H**igher **O**bservational **T**ype **T**heory?

I think **H**igher **O**TT is very promising and it could be an improvement over cubical type theory in many respects once it is finished. But I think it is not what I am looking for.



Why not **H**igher **O**bservational **T**ype Theory?

I think **H**igher **O**TT is very promising and it could be an improvement over cubical type theory in many respects once it is finished. But I think it is not what I am looking for.

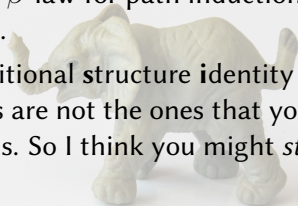
- + I now believe that the definitional β -law for path induction (*i.e.* regularity) is extremely important.



Why not Higher **O**bservational **T**ype Theory?

I think **H**igher **O**TT is very promising and it could be an improvement over cubical type theory in many respects once it is finished. But I think it is not what I am looking for.

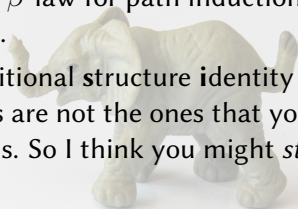
- + I now believe that the definitional β -law for path induction (*i.e.* regularity) is extremely important.
- + Higher OTT provides nearly definitional structure identity principles, but the automatic ones are not the ones that you expect from standard mathematics. So I think you might *still* want to characterise path spaces.



Why not Higher Observational Type Theory?

I think **Higher OTT** is very promising and it could be an improvement over cubical type theory in many respects once it is finished. But I think it is not what I am looking for.

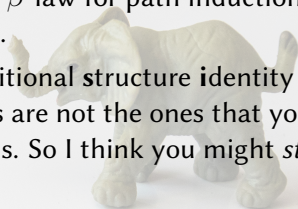
- + I now believe that the definitional β -law for path induction (*i.e.* regularity) is extremely important.
- + Higher OTT provides nearly definitional structure identity principles, but the automatic ones are not the ones that you expect from standard mathematics. So I think you might *still* want to characterise path spaces.
- + Very simple until you realise you need symmetries, after which point I feel we run into usability problems similar to cubical.



Why not Higher Observational Type Theory?

I think **Higher OTT** is very promising and it could be an improvement over cubical type theory in many respects once it is finished. But I think it is not what I am looking for.

- + I now believe that the definitional β -law for path induction (*i.e.* regularity) is extremely important.
- + Higher OTT provides nearly definitional structure identity principles, but the automatic ones are not the ones that you expect from standard mathematics. So I think you might *still* want to characterise path spaces.
- + Very simple until you realise you need symmetries, after which point I feel we run into usability problems similar to cubical.
- + I need something that I know how to implement efficiently (with *complete* higher order pattern unification, *etc.*) today.

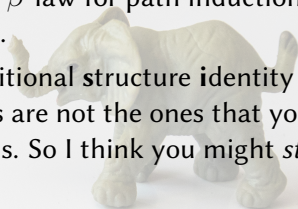


Why not Higher Observational Type Theory?

I think **Higher OTT** is very promising and it could be an improvement over cubical type theory in many respects once it is finished. But I think it is not what I am looking for.

- + I now believe that the definitional β -law for path induction (*i.e.* regularity) is extremely important.
- + Higher OTT provides nearly definitional structure identity principles, but the automatic ones are not the ones that you expect from standard mathematics. So I think you might *still* want to characterise path spaces.
- + Very simple until you realise you need symmetries, after which point I feel we run into usability problems similar to cubical.
- + I need something that I know how to implement efficiently (with *complete* higher order pattern unification, *etc.*) today.

I am **very open** to revisiting these points in the future.





Some proposals for a future proof assistant

“PROJECT PTERODACTYL”



A detailed illustration of a pterosaur, possibly a Pteranodon, in flight. The pterosaur has a long, red, pointed beak and a large, reddish-brown head crest. Its wings are spread wide, showing a yellowish-orange membrane with dark veins. The background features a lush green landscape with a winding river, palm trees, and a large volcano in the distance under a blue sky with light clouds.

Some proposals for a future proof assistant

“PROJECT PTERODACTYL”

(because we always have to
have a bird name)

I want a proof assistant that is...

- + **delightful** for students and pros alike,
- + **predictable** for the educated user,
- + **efficiently** implementable,
- + **orthodox** in its foundational assumptions,
- + and **compatible** with HoTT/UF from the start.

It should be equally usable for constructive *and* classical mathematics.

I want a proof assistant that is...

- + **delightful** for students and pros alike,
- + **predictable** for the educated user,
- + **efficiently** implementable,
- + **orthodox** in its foundational assumptions,
- + and **compatible** with HoTT/UF from the start.

It should be equally usable for constructive *and* classical mathematics.

Major constraint: Interpretable in a conservative extension of Book HoTT. The image may enjoy *more* definitional equalities than are derivable in the source language.

I want a proof assistant that is...

- + **delightful** for students and pros alike,
- + **predictable** for the educated user,
- + **efficiently** implementable,
- + **orthodox** in its foundational assumptions,
- + and **compatible** with HoTT/UF from the start.

It should be equally usable for constructive *and* classical mathematics.

Major constraint: Interpretable in a conservative extension of Book HoTT. The image may enjoy *more* definitional equalities than are derivable in the source language.

Opportunity: to use what we know to make a system that is *even more* predictable and delightful than Lean today, and supports HoTT/UF natively. May our rising tide float all boats!

Dealing with algebraic hierarchies.

The trouble with type classes...

Many systems support some variation on “type classes”, but I think they aren’t conducive to the reliable and predictable organisation of concepts. One issue is what I call the **amnesia problem**:

```
def myTheorem {G: Type} [Group G]: ... G ...
```

The trouble with type classes...

Many systems support some variation on “type classes”, but I think they aren’t conducive to the reliable and predictable organisation of concepts. One issue is what I call the **amnesia problem**:

```
def myTheorem {G: Type} [Group G]: ... G ...
```

Above, wherever G appears after the colon, we know only G : **Type**; if we want to use some of the group operations, we need to consult our type class database to find a way to coerce G to a group.

Easy in this case, but in general relies on **higher-order unification**—for which there do exist well-delineated fragments with **complete** (*i.e.* reliable) algorithms, but these are implemented by no existing proof assistants at all (I am not kidding).

What about bundling?

Why did we not just start with an actual group G : Group and then automatically coerce it to its carrier?

In real mathematics, we say

Let G be a group,

not

Let G be a set, which by the way happens to be the carrier of some group.

What about bundling?

Why did we not just start with an actual group G : Group and then automatically coerce it to its carrier?

In real mathematics, we say

Let G be a group,

not

Let G be a set, which by the way happens to be the carrier of some group.

That's called **bundling**, and this is indeed how the Rocq/**mathcomp** library is organised.

Challenges posed by bundling

1. Need to unbundle to talk about shared components.
2. Rocq/**mathcomp** *still* wants to hang the identity of structures on their carrier sets. (Unification Hell!)

mathcomp walks the narrow path of a verbose and fragile pattern, which is beginning to be automated via **Hierarchy Builder**.

Sometimes get challenging error messages that can only be interpreted by someone who has **implemented** Rocq's unifier.

Shows great promise for Rocq, but I will choose a different design.

Shall we return to first principles? Some theses.

Shall we return to first principles? Some theses.

1. In everyday mathematics, we use a mixture of bundled and unbundled representations—and we pass between them seamlessly. **We must support this directly!**

Shall we return to first principles? Some theses.

1. In everyday mathematics, we use a mixture of bundled and unbundled representations—and we pass between them seamlessly. **We must support this directly!**
2. It is of questionable value to deduce algebraic operations (\times) from type identity (\mathbb{N}) via unification because any given type *does* carry distinct but equally important instances of the same structure. **We should not waste our time trying to do this!**

Shall we return to first principles? Some theses.

1. In everyday mathematics, we use a mixture of bundled and unbundled representations—and we pass between them seamlessly. **We must support this directly!**
2. It is of questionable value to deduce algebraic operations (\times) from type identity (\mathbb{N}) via unification because any given type *does* carry distinct but equally important instances of the same structure. **We should not waste our time trying to do this!**
3. Isabelle deals with all these issues very elegantly and simply, through its *locale* mechanism. **We should adapt aspects of Isabelle's locales to dependent type theory.**

I always found that if you want to make a new programming tool, you need to write the “dream code” first.

“Dream code” is code that (1) you want to be able to write in the final system, and (2) you basically know how to write an elaborator/compiler for.

Here is some of my dream code.

Named telescopes for theory signatures

At first, these are very much like record/structure types in Lean.

theory Semigroup where

car: **Set**

mul: car \rightarrow car \rightarrow car

assoc: (x, y, z: car) \rightarrow mul (mul x y) z = mul x (mul y z)

theory Monoid where

include Semigroup

one: car

leftUnit: (x: car) \rightarrow mul one x = x

rightUnit: (x: car) \rightarrow mul x one = x

Partial instantiation of named telescopes

From **ML modules** and **Arend's records** we take seamless transition between parameterised and bundled structures via convenient *partial instantiation*.

abbreviation $\text{NatMonoid} \equiv \text{Monoid} / \{car \Rightarrow \mathbb{N}\}$

// The result is actually a **small** type! *Semantically* this is just a substitution of a suffix of the telescope.

It is easy to talk about two monoids with the same carrier set.

theory EckmannHilton **where**

X: Set

M, N: Monoid / $\{car \Rightarrow X\}$

dist: $(u, v, w, x: X) \rightarrow M.mul (N.mul\ u\ v) (N.mul\ w\ x) =$
 $N.mul (M.mul\ u\ w) (M.mul\ v\ x)$

Named telescopes are *extensible*

Unlike ML modules and **like Agda and Lean's records**, the name of a telescope matters.

Named telescopes are *extensible*

Unlike ML modules and **like Agda and Lean's records**, the name of a telescope matters. **From Isabelle's locales** we take the ability to retroactively graft *definitional extensions* onto existing theories.

Named telescopes are *extensible*

Unlike ML modules and **like Agda and Lean's records**, the name of a telescope matters. **From Isabelle's locales** we take the ability to retroactively graft *definitional extensions* onto existing theories.

extension Semigroup where

square: car \rightarrow car

square x \Rightarrow mul x x

The **square** function is *henceforth* available on not only any **Semigroup**, but also any **Monoid**, or any **NatMonoid**, *etc.*

Named telescopes are *extensible*

Unlike ML modules and like Agda and Lean's records, the name of a telescope matters. From Isabelle's *locales* we take the ability to retroactively graft *definitional extensions* onto existing theories.

extension Semigroup **where**

square: car \rightarrow car

square x \Rightarrow mul x x

The **square** function is *henceforth* available on not only any **Semigroup**, but also any **Monoid**, or any **NatMonoid**, *etc.*

extension EckmannHilton **where**

result: M = N

result \Rightarrow // ... proof of Eckmann–Hilton theorem!

Idea: set up a theory whose fields are the assumptions of a complicated result, and extend it by the result.

Compared with type classes

Extensible named telescopes, which combine aspects of ML modules and Isabelle's locales, are a lightweight alternative to type classes (without covering all the latter's features).

Compared with type classes

Extensible named telescopes, which combine aspects of ML modules and Isabelle's locales, are a lightweight alternative to type classes (without covering all the latter's features).

Extensible named telescopes do their heavy lifting **without resorting to higher-order unification**; this makes them both highly efficient *and* predictable.

No free lunch: unification still crucial, but it need not steer the ship.

Compared with type classes

Extensible named telescopes, which combine aspects of ML modules and Isabelle's locales, are a lightweight alternative to type classes (without covering all the latter's features).

Extensible named telescopes do their heavy lifting **without resorting to higher-order unification**; this makes them both highly efficient *and* predictable.

No free lunch: unification still crucial, but it need not steer the ship.

Diamond problems are easily resolved, both in the sense of resolution blowup *and* in the far more important sense of coherence. (Lean's diamond resolution is fast but incoherent and unspecified.)

Even cycles are fine. (Isabelle shows the way!)

Dealing with inductive types and eliminators.

How shall we deal with inductive types?

Several theses.

How shall we deal with inductive types?

Several theses.

1. I propose to follow **Epigram** in providing pattern notation to **anything that looks like an eliminator**. Can be adapted to HoTT after Cockx, Devriese, and Piessens.

How shall we deal with inductive types?

Several theses.

1. I propose to follow **Epigram** in providing pattern notation to **anything that looks like an eliminator**. Can be adapted to HoTT after Cockx, Devriese, and Piessens.
2. My goal in eliminating pattern matching is not only trust: **non-built-in eliminators deserve good notation too**.

How shall we deal with inductive types?

Several theses.

1. I propose to follow **Epigram** in providing pattern notation to **anything that looks like an eliminator**. Can be adapted to HoTT after Cockx, Devriese, and Piessens.
2. My goal in eliminating pattern matching is not only trust: **non-built-in eliminators deserve good notation too**.
3. There is **no need for fancy termination checkers**; even Agda's non-fancy termination checker has been a fount of inconsistency. **Mouse + Cookie problem**.

How shall we deal with inductive types?

Several theses.

1. I propose to follow **Epigram** in providing pattern notation to **anything that looks like an eliminator**. Can be adapted to HoTT after Cockx, Devriese, and Piessens.
2. My goal in eliminating pattern matching is not only trust: **non-built-in eliminators deserve good notation too**.
3. There is **no need for fancy termination checkers**; even Agda's non-fancy termination checker has been a fount of inconsistency. **Mouse + Cookie problem**.
4. (Higher) inductive type declarations can be **elaborated to signatures** from which eliminators can be synthesised à la Kaposi-Kovacs.

How shall we deal with inductive types?

Several theses.

1. I propose to follow **Epigram** in providing pattern notation to **anything that looks like an eliminator**. Can be adapted to HoTT after Cockx, Devriese, and Piessens.
2. My goal in eliminating pattern matching is not only trust: **non-built-in eliminators deserve good notation too**.
3. There is **no need for fancy termination checkers**; even Agda's non-fancy termination checker has been a fount of inconsistency. **Mouse + Cookie problem**.
4. (Higher) inductive type declarations can be **elaborated to signatures** from which eliminators can be synthesised à la Kaposi-Kovacs.
5. **Usability**: the user should not see the unravelled signatures *nor* eliminators, ever(*). This will work *even* in the case of user-supplied eliminators.

Why haven't we been doing for the twenty years?

Basing a proof assistant on eliminators is widely believed to have some drawbacks.

Why haven't we been doing for the twenty years?

Basing a proof assistant on eliminators is widely believed to have some drawbacks.

1. **Writeability:** Eliminators are difficult to program directly with.

Why haven't we been doing for the twenty years?

Basing a proof assistant on eliminators is widely believed to have some drawbacks.

1. **Writeability:** Eliminators are difficult to program directly with.
Fact Check: Conor McBride solved this in their 1999 PhD thesis, inspiring modern systems like Lean to compile pattern matching to eliminators.

Why haven't we been doing for the twenty years?

Basing a proof assistant on eliminators is widely believed to have some drawbacks.

1. **Writeability:** Eliminators are difficult to program directly with.
Fact Check: Conor McBride solved this in their 1999 PhD thesis, inspiring modern systems like Lean to compile pattern matching to eliminators.
2. **Readability:** If the kernel uses eliminators, they are going to show up in goals, which makes things impossible to read.

Why haven't we been doing for the twenty years?

Basing a proof assistant on eliminators is widely believed to have some drawbacks.

1. **Writeability:** Eliminators are difficult to program directly with.
Fact Check: Conor McBride solved this in their 1999 PhD thesis, inspiring modern systems like Lean to compile pattern matching to eliminators.
2. **Readability:** If the kernel uses eliminators, they are going to show up in goals, which makes things impossible to read.
Fact Check: See McBride and McKinna, *The View From The Left*. This is a solved problem, using McBride's "labelled types" trick!

Why haven't we been doing for the twenty years?

Basing a proof assistant on eliminators is widely believed to have some drawbacks.

1. **Writeability:** Eliminators are difficult to program directly with.
Fact Check: Conor McBride solved this in their 1999 PhD thesis, inspiring modern systems like Lean to compile pattern matching to eliminators.
2. **Readability:** If the kernel uses eliminators, they are going to show up in goals, which makes things impossible to read.
Fact Check: See McBride and McKinna, *The View From The Left*. This is a solved problem, using McBride's "labelled types" trick!
3. **Efficiency:** Code written with eliminators is less efficient than general recursive algorithms.

Why haven't we been doing for the twenty years?

Basing a proof assistant on eliminators is widely believed to have some drawbacks.

1. **Writeability:** Eliminators are difficult to program directly with.
Fact Check: Conor McBride solved this in their 1999 PhD thesis, inspiring modern systems like Lean to compile pattern matching to eliminators.
2. **Readability:** If the kernel uses eliminators, they are going to show up in goals, which makes things impossible to read.
Fact Check: See McBride and McKinna, *The View From The Left*. This is a solved problem, using McBride's "labelled types" trick!
3. **Efficiency:** Code written with eliminators is less efficient than general recursive algorithms.
Fact Check: See Brady, *Practical Implementation of a Dependently Typed Functional Programming Language* (§ 6.1.3). Sometimes true sometimes not, but Brady shows how to recover the intended general recursive algorithm if we need it using labelled types.

Why haven't we been doing for the twenty years?

Basing a proof assistant on eliminators is widely believed to have some drawbacks.

1. **Writeability:** Eliminators are difficult to program directly with.
Fact Check: Conor McBride solved this in their 1999 PhD thesis, inspiring modern systems like Lean to compile pattern matching to eliminators.
2. **Readability:** If the kernel uses eliminators, they are going to show up in goals, which makes things impossible to read.
Fact Check: See McBride and McKinna, *The View From The Left*. This is a solved problem, using McBride's "labelled types" trick!
3. **Efficiency:** Code written with eliminators is less efficient than general recursive algorithms.
Fact Check: See Brady, *Practical Implementation of a Dependently Typed Functional Programming Language* (§ 6.1.3). Sometimes true sometimes not, but Brady shows how to recover the intended general recursive algorithm if we need it using labelled types.
4. **Expressivity:** It might be very difficult to interpret reasonable but subtle total recursive programs.

Why haven't we been doing for the twenty years?

Basing a proof assistant on eliminators is widely believed to have some drawbacks.

1. **Writeability:** Eliminators are difficult to program directly with.
Fact Check: Conor McBride solved this in their 1999 PhD thesis, inspiring modern systems like Lean to compile pattern matching to eliminators.
2. **Readability:** If the kernel uses eliminators, they are going to show up in goals, which makes things impossible to read.
Fact Check: See McBride and McKinna, *The View From The Left*. This is a solved problem, using McBride's "labelled types" trick!
3. **Efficiency:** Code written with eliminators is less efficient than general recursive algorithms.
Fact Check: See Brady, *Practical Implementation of a Dependently Typed Functional Programming Language* (§ 6.1.3). Sometimes true sometimes not, but Brady shows how to recover the intended general recursive algorithm if we need it using labelled types.
4. **Expressivity:** It might be very difficult to interpret reasonable but subtle total recursive programs.
Fact Check: Those patterns are *not* reasonable, and we *shouldn't* support them. (See the last 900 soundness bugs in Agda.) Nonetheless, we can arrange to *compile* our eliminator code to **any** subtle general-recursive program we want (using labelled types, again).

Current art in the elimination of patterns

- + **Lean** compiles recursive pattern-matching programs to eliminators. Support for *dependent* pattern matching is a bit limited. *Works very well and constantly improving.*

Current art in the elimination of patterns

- + **Lean** compiles recursive pattern-matching programs to eliminators. Support for *dependent* pattern matching is a bit limited. *Works very well and constantly improving.*
- + **Rocq** has the **Equations** package by Sozeau, which implements more of McBride and McKinna's vision than Lean, but a little rough around the edges.

Current art in the elimination of patterns

- + **Lean** compiles recursive pattern-matching programs to eliminators. Support for *dependent* pattern matching is a bit limited. *Works very well and constantly improving.*
- + **Rocq** has the **Equations** package by Sozeau, which implements more of McBride and McKinna's vision than Lean, but a little rough around the edges.
- + Both Lean and Rocq/Equations use *accessibility predicates* in their elaboration of structural recursion.
 - Lean's accessibility predicates are proof irrelevant, which destroys subject reduction. Oops!
 - (I'll say more about this.)

A few lessons our community never learned...

In their PhD (1999), Conor McBride sought to elaborate dependent pattern matching to a data type's standard eliminator, leading to **Epigram**. Basis for Lean's and Rocq/Equations' pattern matching. **Along the way, a few lessons of Epigram were missed.**

A few lessons our community never learned...

In their PhD (1999), Conor McBride sought to elaborate dependent pattern matching to a data type's standard eliminator, leading to **Epigram**. Basis for Lean's and Rocq/Equations' pattern matching. **Along the way, a few lessons of Epigram were missed.**

1. Epigram's pattern matching notation applies to *anything* that is shaped like an eliminator. **HoTT/UF applications abound.**

A few lessons our community never learned...

In their PhD (1999), Conor McBride sought to elaborate dependent pattern matching to a data type's standard eliminator, leading to **Epigram**. Basis for Lean's and Rocq/Equations' pattern matching. **Along the way, a few lessons of Epigram were missed.**

1. Epigram's pattern matching notation applies to *anything* that is shaped like an eliminator. **HoTT/UF applications abound.**
2. Epigram did *not* force accessibility witnesses: instead, used “memo structures” inspired by Giménez. **More to say.**

It it looks like and quacks like an eliminator...

Conor McBride gave the following example:

Given

$\text{vproj}: \text{Vec } X \ n \rightarrow \mathbf{Fin} \ n \rightarrow X$

how would you define the following?

$\text{vproj?}: \text{Vec } X \ n \rightarrow \mathbb{N} \rightarrow \text{Maybe } X$

It it looks like and quacks like an eliminator...

Conor McBride gave the following example:

Given

$\text{vproj}: \text{Vec } X \ n \rightarrow \mathbf{Fin} \ n \rightarrow X$

how would you define the following?

$\text{vproj}?: \text{Vec } X \ n \rightarrow \mathbb{N} \rightarrow \text{Maybe } X$

We could first project the vector to a list, but that is inefficient. Instead we want to check the bound, but we need to do this in a very dependently typed way.

Conor proposes to use a *derived eliminator* that justifies the case split that we want to perform in general:

$$\pi: \text{Fin } n \rightarrow \mathbb{N}$$

checkBound

$$\begin{aligned} &: (P: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbf{Type}) \\ &\rightarrow ((n: \mathbb{N}) (i: \mathbf{Fin } n) \rightarrow P \, n \, (\pi i)) \\ &\rightarrow ((n, r: \mathbb{N}) \rightarrow P \, n \, (n + r)) \\ &\rightarrow (n, m: \mathbb{N}) \rightarrow P \, n \, m \end{aligned}$$

// prove this later

Now we can do dependent pattern matching, but using `checkBound` instead of the standard eliminator.

Now we can do dependent pattern matching, but using `checkBound` instead of the standard eliminator. Here's roughly how it looked in Epigram:

$$\text{vproj?}_n (\vec{x}: \text{Vec } X \ n) (m: \mathbb{N}) \rightarrow \text{Maybe } X$$
$$\text{vproj?}_n \vec{x} \ m$$

Now we can do dependent pattern matching, but using `checkBound` instead of the standard eliminator. Here's roughly how it looked in Epigram:

```
vproj?_n (x̄: Vec X n) (m: ℕ) → Maybe X  
vproj?_n x̄ m ⇐ checkBound n m  
vproj?_n x̄ (π i)  
vproj?_n x̄ (n + m)
```

Now we can do dependent pattern matching, but using `checkBound` instead of the standard eliminator. Here's roughly how it looked in Epigram:

```
vproj?_n (x̄: Vec X n) (m: ℕ) → Maybe X  
vproj?_n x̄ m ⇐ checkBound n m  
vproj?_n x̄ (π i) ⇒ some (vproj x̄ i)  
vproj?_n x̄ (n + m)
```


Now we can do dependent pattern matching, but using `checkBound` instead of the standard eliminator. Here's roughly how it looked in Epigram:

```
vproj?_n (x̄: Vec X n) (m: ℕ) → Maybe X  
vproj?_n x̄ m ⇐ checkBound n m  
vproj?_n x̄ (π i) ⇒ some (vproj x̄ i)  
vproj?_n x̄ (n + m) ⇒ none
```



Now we can do dependent pattern matching, but using `checkBound` instead of the standard eliminator. Here's roughly how it looked in Epigram:

```
vproj?_n (x̄: Vec X n) (m: ℕ) → Maybe X
vproj?_n x̄ m ⇐ checkBound n m
vproj?_n x̄ (π i) ⇒ some (vproj x̄ i)
vproj?_n x̄ (n + m) ⇒ none
```



We can simulate this in Agda using several more indirections, **where**-clauses, *etc.*. But (1) directness is good, and (2) the recipe of McBride and McKinna gives better control over the display of goals.

Derived eliminators are useful for HoTT/UF!

Consider the set-quotient $[-]: A \twoheadrightarrow A/R$. We have the standard eliminator:

$$\begin{aligned} \text{elim}_{A/R} &: (B : A/R \rightarrow \mathbf{Set}) \\ &\rightarrow (f : (x : A) \rightarrow B [x]) \\ &\rightarrow (x, y : A) (r : R \ x \ y) \rightarrow fx =_{\text{glue } r}^B fy \\ &\rightarrow (x : A/R) \rightarrow B \ x \end{aligned}$$

Derived eliminators are useful for HoTT/UF!

Consider the set-quotient $[-]: A \twoheadrightarrow A/R$. We have the standard eliminator:

$$\begin{aligned} \text{elim}_{A/R} & \\ &: (B : A/R \rightarrow \mathbf{Set}) \\ &\rightarrow (f : (x : A) \rightarrow B [x]) \\ &\rightarrow (x, y : A) (r : R \ x \ y) \rightarrow fx =_{\text{glue } r}^B fy \\ &\rightarrow (x : A/R) \rightarrow B \ x \end{aligned}$$

But we can also *derive* an eliminator for proposition-valued motives!

$$\begin{aligned} \text{elimProp}_{A/R} & \\ &: (B : A/R \rightarrow \mathbf{Prop}) \\ &\rightarrow (f : (x : A) \rightarrow B [x]) \\ &\rightarrow (u : A/R) \rightarrow B \ u \end{aligned}$$

This would allow us to give the following very elegant proof that the quotient map is a surjection:

$\text{quotientMapIsSurj } (u: A/R): \exists x: A. u = [x]$

$\text{quotientMapIsSurj } u$

This would allow us to give the following very elegant proof that the quotient map is a surjection:


```
quotientMapIsSurj (u: A/R): ∃x: A. u = [x]  
quotientMapIsSurj u ⇐ elimPropA/R u  
quotientMapIsSurj [x]
```

This would allow us to give the following very elegant proof that the quotient map is a surjection:

$\text{quotientMapIsSurj } (u: A/R): \exists x: A. u = [x]$
 $\text{quotientMapIsSurj } u \Leftarrow \text{elimProp}_{A/R} u$
 $\text{quotientMapIsSurj } [x] \Rightarrow |(x, \text{refl})|_{-1}$



This would allow us to give the following very elegant proof that the quotient map is a surjection:

$$\begin{aligned} \text{quotientMapIsSurj } (u: A/R) &: \exists x: A. u = [x] \\ \text{quotientMapIsSurj } u &\Leftarrow \text{elimProp}_{A/R} u \\ \text{quotientMapIsSurj } [x] &\Rightarrow |(x, \text{refl})|_{-1} \end{aligned}$$


Unfolding behaviour is extremely good: the user never sees

$$\text{elimProp}_{A/R} u (\lambda x. |(x, \text{refl})|_{-1})$$


in a goal, but nonetheless the unfolding

$$\text{quotientMapIsSurj } [x] \rightsquigarrow |(x, \text{refl})|_{-1}$$

does fire.

This would allow us to give the following very elegant proof that the quotient map is a surjection:

```
quotientMapIsSurj (u: A/R): ∃x: A. u = [x]
quotientMapIsSurj u ⇐ elimPropA/R u
quotientMapIsSurj [x] ⇒ |(x, refl)|-1
```



Unfolding behaviour is extremely good: the user never sees

$$\text{elimProp}_{A/R} u (\lambda x. |(x, \text{refl})|_{-1})$$

in a goal, but nonetheless the unfolding

$$\text{quotientMapIsSurj } [x] \rightsquigarrow |(x, \text{refl})|_{-1}$$

does fire. **This is automatic and works for *anything* shaped like an eliminator.** Thanks, Conor!

Two derived eliminators for HoTT/UF

Both function extensionality and the univalence axiom can be rephrased as *induction principles*.

1. Function extensionality = “induction on homotopies”
2. Univalence = “induction on equivalences”

More generally for any *univalent reflexive graph* we obtain a derived eliminator (see “fundamental theorem of identity types”).

$\text{cor2.4.4}_f: A \rightarrow A \ (H: f \sim 1_A): H \circ f \sim \text{ap}_f \circ H$
 $\text{cor2.4.4}_f \ H \ p \ x$

$\text{cor2.4.4}_f: A \rightarrow A \ (H: f \sim 1_A): H \circ f \sim \text{ap}_f \circ H$
 $\text{cor2.4.4}_f \ H \ p \ x \Leftarrow \text{htpyInduction} \ (f, H)$
 $\text{cor2.4.4}_{1_A} \ (\lambda x. \text{refl}_x)$

$\text{cor2.4.4}_f: A \rightarrow A \ (H: f \sim 1_A): H \circ f \sim \text{ap}_f \circ H$
 $\text{cor2.4.4}_f \ H \ p \ x \Leftarrow \text{htpyInduction} \ (f, H)$
 $\text{cor2.4.4}_{1_A} \ (\lambda x. \text{refl}_x) \Rightarrow \lambda x. \text{refl}_{\text{refl}_x}$



$$\begin{aligned} & \text{equivCompAssoc}_{A,B,C,D} (f : A \cong B) (g : B \cong C) (h : C \cong D) \\ & : h \circ (g \circ f) =_{A \cong C} (h \circ g) \circ f \\ & \text{equivCompAssoc}_{A,B,C,D} f g h \end{aligned}$$

$\text{equivCompAssoc}_{A,B,C,D} (f : A \cong B) (g : B \cong C) (h : C \cong D)$

$: h \circ (g \circ f) =_{A \cong C} (h \circ g) \circ f$

$\text{equivCompAssoc}_{A,B,C,D} f g h \Leftarrow \text{equivInduction } f$

$\text{equivCompAssoc}_{A,A,C,D} 1_A g h$

$$\text{equivCompAssoc}_{A,B,C,D} (f : A \cong B) (g : B \cong C) (h : C \cong D) \\ : h \circ (g \circ f) =_{A \cong C} (h \circ g) \circ f$$

$$\text{equivCompAssoc}_{A,B,C,D} f g h \Leftarrow \text{equivInduction } f$$

$$\text{equivCompAssoc}_{A,A,C,D} 1_A g h \Leftarrow \text{equivInduction } g$$

$$\text{equivCompAssoc}_{B,B,B,D} 1_A 1_A h$$

$$\text{equivCompAssoc}_{A,B,C,D} (f : A \cong B) (g : B \cong C) (h : C \cong D) \\ : h \circ (g \circ f) =_{A \cong C} (h \circ g) \circ f$$

$$\text{equivCompAssoc}_{A,B,C,D} f g h \Leftarrow \text{equivInduction } f$$

$$\text{equivCompAssoc}_{A,A,C,D} 1_A g h \Leftarrow \text{equivInduction } g$$

$$\text{equivCompAssoc}_{B,B,B,D} 1_A 1_A h \Leftarrow \text{equivInduction } h$$

$$\text{equivCompAssoc}_{B,B,B,1_A} 1_A 1_A 1_A$$

$$\text{equivCompAssoc}_{A,B,C,D} (f : A \cong B) (g : B \cong C) (h : C \cong D) \\ : h \circ (g \circ f) =_{A \cong C} (h \circ g) \circ f$$

$$\text{equivCompAssoc}_{A,B,C,D} f g h \Leftarrow \text{equivInduction } f$$

$$\text{equivCompAssoc}_{A,A,C,D} 1_A g h \Leftarrow \text{equivInduction } g$$

$$\text{equivCompAssoc}_{B,B,B,D} 1_A 1_A h \Leftarrow \text{equivInduction } h$$

$$\text{equivCompAssoc}_{B,B,B,1_A} 1_A 1_A 1_A \Rightarrow \text{refl}$$



What about “typal” computation rules?

We need a notation for the path constructor cases when matching on HITs. You could something like

$$\text{apd } f \text{ loop} \Rightarrow \{ \dots \}$$

In Book HoTT, this “case” does not correspond to a definitional equality. **Important! Not just a defect of Book HoTT.**

Epigram’s elimination of pattern matching works regardless of whether there is any computational behaviour associated to the eliminator whatsoever.

Promised remarks on accessibility predicates

Lean and Rocq/Equations elaborate structural recursion in terms of **accessibility predicates**.

Promised remarks on accessibility predicates

Lean and Rocq/Equations elaborate structural recursion in terms of **accessibility predicates**.

data $\text{Acc} (R: A \rightarrow A \rightarrow \mathbf{Type}) (x: A): \mathbf{Type}$ **where**

$\text{acc} : \left(\prod_{y:A} R y x \rightarrow \text{Acc } y \right) \rightarrow \text{Acc } x$

$\mathbb{N}.\text{acc} : \prod_{x:\mathbb{N}} \text{Acc } (<) x$

Promised remarks on accessibility predicates

Lean and Rocq/Equations elaborate structural recursion in terms of **accessibility predicates**.

```
data Acc (R: A → A → Type) (x: A): Type where  
  acc : (∏y:A Ryx → Acc y) → Acc x
```

$$\mathbb{N}.\text{acc}: \prod_{x:\mathbb{N}} \text{Acc} (<) x$$

Then a function $f: \prod_{x:\mathbb{N}} B[x]$ is elaborated to like so:

```
 $\hat{f}: \prod_{x:\mathbb{N}} \text{Acc} (<) x \rightarrow B[x]$   
// implement by induction on Acc argument!
```

$$fx \equiv \hat{f} \ x \ (\mathbb{N}.\text{acc} \ x)$$

This works fine. Recursive calls must pass a proof that their target is strictly smaller than the current case.

This works fine. Recursive calls must pass a proof that their target is strictly smaller than the current case.

Subtlety: Elaborated function calls `Acc`'s eliminator on its accessibility argument. Computation is stuck unless the argument is of the form “`acc h`” (important!).

This works fine. Recursive calls must pass a proof that their target is strictly smaller than the current case.

Subtlety: Elaborated function calls Acc 's eliminator on its accessibility argument. Computation is stuck unless the argument is of the form “ $\text{acc } h$ ” (important!).

Consequence: Making Acc proof-irrelevant immediately leads to undecidability, because under proof irrelevance, if *any* such h exists, then any proof $\phi: \text{Acc } (<) x$ is definitionally equal to $\text{acc } h$.

This works fine. Recursive calls must pass a proof that their target is strictly smaller than the current case.

Subtlety: Elaborated function calls `Acc`'s eliminator on its accessibility argument. Computation is stuck unless the argument is of the form “`acc h`” (important!).

Consequence: Making `Acc` proof-irrelevant immediately leads to undecidability, because under proof irrelevance, if *any* such `h` exists, then any proof $\phi : \text{Acc } (<) x$ is definitionally equal to `acc h`.

So much gnashing of teeth, but remember:

Doctor, it hurts when I [make Acc proof-irrelevant]...

Then stop [making Acc proof-irrelevant]!

...

Giménez's memo structures as a “family transformer”

McBride proposed an alternative approach that avoids forcing accessibility witnesses, inspired by Giménez.

Giménez's memo structures as a “family transformer”

McBride proposed an alternative approach that avoids forcing accessibility witnesses, inspired by Giménez.

Instead of pattern matching on the accessibility proof, just take *all possible structural recursive calls* as an argument (deep i.h.).

$$\mathbb{N}.\text{Memo}_P: \mathbb{N} \rightarrow \mathbf{Type} : \mathbb{N} \rightarrow \mathbf{Type}$$
$$\mathbb{N}.\text{Memo}_P\ 0 \equiv \mathbf{1}$$
$$\mathbb{N}.\text{Memo}_P\ (n + 1) \equiv P[n] \times \mathbb{N}.\text{Memo}_P\ n$$

Giménez's memo structures as a “family transformer”

McBride proposed an alternative approach that avoids forcing accessibility witnesses, inspired by Giménez.

Instead of pattern matching on the accessibility proof, just take *all possible structural recursive calls* as an argument (deep i.h.).

$$\mathbb{N}.\text{Memo}_P : \mathbb{N} \rightarrow \mathbf{Type} : \mathbb{N} \rightarrow \mathbf{Type}$$

$$\mathbb{N}.\text{Memo}_P 0 \equiv \mathbf{1}$$

$$\mathbb{N}.\text{Memo}_P (n + 1) \equiv P[n] \times \mathbb{N}.\text{Memo}_P n$$

Then functions $f : \prod_{x:\mathbb{N}} P[x]$ are elaborated to strengthen the i.h. via:

$$\mathbb{N}.\text{rec}_P : (\prod_{x:\mathbb{N}} \mathbb{N}.\text{Memo}_P x \rightarrow P[x]) \rightarrow \prod_{x:\mathbb{N}} P[x]$$

Accessibility witnesses vs. memo structures

Accessibility witnesses vs. memo structures

Mathematical insight: the two approaches are equivalent, basically because **the mapping-out space of a colimit is a limit.**

Accessibility witnesses vs. memo structures

Mathematical insight: the two approaches are equivalent, basically because **the mapping-out space of a colimit is a limit.**

But limits always work better than colimits in type theory!

Accessibility witnesses vs. memo structures

Mathematical insight: the two approaches are equivalent, basically because **the mapping-out space of a colimit is a limit**.

But limits always work better than colimits in type theory!

Operational insight: The accessibility version must **force** its auxiliary argument, whereas the memo structure version is **lazy** in its auxiliary argument.

Accessibility witnesses vs. memo structures

Mathematical insight: the two approaches are equivalent, basically because **the mapping-out space of a colimit is a limit**.

But limits always work better than colimits in type theory!

Operational insight: The accessibility version must **force** its auxiliary argument, whereas the memo structure version is **lazy** in its auxiliary argument.

Memo structures = a more type-theoretically friendly interpretation that entirely avoids the question of proof-irrelevance.

Combined with **labelled types**, readability of goals is assured.

Dealing with editors.

Moving on from batch mode

I have always built proof assistants in “batch mode”. (JonPRL, RedPRL, redtt, cooltt.)

Moving on from batch mode

I have always built proof assistants in “batch mode”. (JonPRL, RedPRL, redtt, cooltt.)

It is better to think about the proof assistant as an interactive server rather than as a compiler.

1. **Nuprl** was literally a Smalltalk-style object database!
2. **Rocq** and **Narya** have an “interaction mode” that can be wired up to **Proof General**.
3. **Lean** has a **language server** that provides advanced IDE-like features to tons of editors that support Microsoft’s **language server protocol**.

Lean’s approach seems to be similar to that of many modern industrial programming languages, like Rust and Swift.

Designing a proof assistant around its language server seemed backward to me at first. **I was wrong.**

Designing a proof assistant around its language server seemed backward to me at first. **I was wrong.**

- + It allows/forces us to design and orchestrate **user interactions** from the start.

Designing a proof assistant around its language server seemed backward to me at first. **I was wrong.**

- + It allows/forces us to design and orchestrate **user interactions** from the start.
- + From Day One, we can have **best-effort editor plugins**, semantic highlighting, *etc.*

Designing a proof assistant around its language server seemed backward to me at first. **I was wrong.**

- + It allows/forces us to design and orchestrate **user interactions** from the start.
- + From Day One, we can have **best-effort editor plugins**, semantic highlighting, *etc.*
- + A reasonable path toward ***incremental elaboration*** for responsiveness.

Designing a proof assistant around its language server seemed backward to me at first. **I was wrong.**

- + It allows/forces us to design and orchestrate **user interactions** from the start.
- + From Day One, we can have **best-effort editor plugins**, semantic highlighting, *etc.*
- + A reasonable path toward ***incremental elaboration*** for responsiveness.
- + Many aspects of the tool architecture have to be *radically* inverted when moving from batch-mode to interaction-mode.
Surrender to the inevitable and take interactivity seriously!

Designing a proof assistant around its language server seemed backward to me at first. **I was wrong.**

- + It allows/forces us to design and orchestrate **user interactions** from the start.
- + From Day One, we can have **best-effort editor plugins**, semantic highlighting, *etc.*
- + A reasonable path toward ***incremental elaboration*** for responsiveness.
- + Many aspects of the tool architecture have to be *radically* inverted when moving from batch-mode to interaction-mode.
Surrender to the inevitable and take interactivity seriously!

When I started, none of this was a user expectation. **Times have changed, we must adapt.**

Thoughts on prevalent language servers

I love *many* aspects of Lean, but I don't love how eager it is to display error diagnostics whilst I'm in the middle of typing something, or how the goals disappear whilst I am preparing a solution.

Thoughts on prevalent language servers

I love *many* aspects of Lean, but I don't love how eager it is to display error diagnostics whilst I'm in the middle of typing something, or how the goals disappear whilst I am preparing a solution.

Using a language server architecture **does not** mean we are required to copy every aspect of Lean, *e.g.* conflating holes with metavariables.

Thoughts on prevalent language servers

I love *many* aspects of Lean, but I don't love how eager it is to display error diagnostics whilst I'm in the middle of typing something, or how the goals disappear whilst I am preparing a solution.

Using a language server architecture **does not** mean we are required to copy every aspect of Lean, *e.g.* conflating holes with metavariables.

We should take the (many) good aspects that we can, and then creatively consider the needs of **proof engineers** and **students** for the rest.

In summary...

I am proposing to build a new proof assistant for HoTT/UF.

In summary...

I am proposing to build a new proof assistant for HoTT/UF.

1. **No experimental type theory!** Orthodoxy is required to make contact with mathematics, *and* to ensure viable implementation strategies for reliable unification, automation, *etc.*

In summary...

I am proposing to build a new proof assistant for HoTT/UF.

1. **No experimental type theory!** Orthodoxy is required to make contact with mathematics, *and* to ensure viable implementation strategies for reliable unification, automation, *etc.*
2. **Simple, reliable, and flexible organisation of algebraic hierarchies** out of the box, combining ideas from Isabelle's locales and ML modules.

In summary...

I am proposing to build a new proof assistant for HoTT/UF.

1. **No experimental type theory!** Orthodoxy is required to make contact with mathematics, *and* to ensure viable implementation strategies for reliable unification, automation, *etc.*
2. **Simple, reliable, and flexible organisation of algebraic hierarchies** out of the box, combining ideas from Isabelle's locales and ML modules.
3. **Usable programming with derived eliminators** inspired by Epigram, adapted for HoTT/UF.

In summary...

I am proposing to build a new proof assistant for HoTT/UF.

1. **No experimental type theory!** Orthodoxy is required to make contact with mathematics, *and* to ensure viable implementation strategies for reliable unification, automation, *etc.*
2. **Simple, reliable, and flexible organisation of algebraic hierarchies** out of the box, combining ideas from Isabelle's locales and ML modules.
3. **Usable programming with derived eliminators** inspired by Epigram, adapted for HoTT/UF.
4. **Fine-grained control over unfolding** both for display *and* abstraction.

In summary...

I am proposing to build a new proof assistant for HoTT/UF.

1. **No experimental type theory!** Orthodoxy is required to make contact with mathematics, *and* to ensure viable implementation strategies for reliable unification, automation, *etc.*
2. **Simple, reliable, and flexible organisation of algebraic hierarchies** out of the box, combining ideas from Isabelle's locales and ML modules.
3. **Usable programming with derived eliminators** inspired by Epigram, adapted for HoTT/UF.
4. **Fine-grained control over unfolding** both for display *and* abstraction.
5. **Rich editor experience from the start.**


In summary...

I am proposing to build a new proof assistant for HoTT/UF.

1. **No experimental type theory!** Orthodoxy is required to make contact with mathematics, *and* to ensure viable implementation strategies for reliable unification, automation, *etc.*
2. **Simple, reliable, and flexible organisation of algebraic hierarchies** out of the box, combining ideas from Isabelle's locales and ML modules.
3. **Usable programming with derived eliminators** inspired by Epigram, adapted for HoTT/UF.
4. **Fine-grained control over unfolding** both for display *and* abstraction.
5. **Rich editor experience from the start.**

This is admittedly a huge project.

One more thing...

A detailed illustration of a pterosaur, possibly a Pteranodon, in flight. The creature has a long, sharp beak, a prominent crest on its head, and large, yellowish-brown wings with dark red or orange markings. It is shown from a side profile, flying towards the left. The background is a soft-focus landscape with green hills and a blue sky with light clouds. The text "I have a prototype (of some of this)..." is overlaid in the center of the image.

I have a prototype (of some of this)...

