# RedPRL

designing the people's refinement logic

Jon Sterling

Carnegie Mellon University

# what is RedPRL?

A project to build a modernized Nuprl for Computational Cubical Type Theory (Angiuli, Harper, Wilson): the first ever ***interactive*** proof assistant for higher dimensional type theory.

# what is RedPRL?

A project to build a modernized Nuprl for Computational Cubical Type Theory (Angiuli, Harper, Wilson): the first ever ***interactive*** proof assistant for higher dimensional type theory.

I hate writing code, and mechanization with current tools frustrates me. I wish everything could be done on paper.

# what is RedPRL?

A project to build a modernized Nuprl for Computational Cubical Type Theory (Angiuli, Harper, Wilson): the first ever ***interactive*** proof assistant for higher dimensional type theory.

I hate writing code, and mechanization with current tools frustrates me. I wish everything could be done on paper.

So, why bother?

# the absolute idea

$$
\begin{tikzcd}
\text{Practical Idea} \arrow[r, bend right, "\textit{theory}"'] \arrow[r, phantom, "\perp"] & \text{Theoretical Idea} \arrow[l, bend right, "\textit{praxis}"']
\end{tikzcd}
$$

(With thanks to Hegel, Marx, Mao and Lawvere.)

# overview of RedPRL

Cubical Refinement Logic

Cubical Abstract Machine

Cubical Type Theory

Cubical Abstract Binding Trees

Dependent LCF

Indexed Second-Order Algebra

Lawvere Duality

# outline

1. The LCF/PRL Tradition
2. The Revisionists And Their Running Dogs
3. RedPRL: New Synthesis Of Proof Refinement

# lcf architecture

```
type evd                                       (* evidence *)
type α state = α list ⊗ (evd list → evd)       (* proof state *)
type (α, β) tactic = α ⇀ β state
```

# lcf architecture

```
type evd                                          (* evidence *)
type α state = α list ⊗ (evd list → evd)          (* proof state *)
type (α, β) tactic = α ⇀ β state

(* proof state “monad” *)
val id : (α, α) tactic
val map : (α → β) → (α state, β) tactic
val mul : (α state state, α) tactic
val orelse : (α, β) tactic ⊗ (α, β) tactic → (α, β) tactic
```

# lcf architecture

```
type evd                                          (* evidence *)
type α state = α list ⊗ (evd list → evd)          (* proof state *)
type (α, β) tactic = α ⇀ β state

(* proof state “monad” *)
val id : (α, α) tactic
val map : (α → β) → (α state, β) tactic
val mul : (α state state, α) tactic
val orelse : (α, β) tactic ⊗ (α, β) tactic → (α, β) tactic

(* standard tacticals *)
val then : (α, β) tactic ⊗ (β, γ) tactic → (α, γ) tactic
fun then (t1, t2) = mul ∘ map t2 ∘ t1
val thenl : (α, β) tactic ⊗ (β, γ) tactic list → (α, γ) tactic
```

# lcf architecture

```
include LCF
```

# lcf architecture

```
include LCF
datatype prop =
  | ∧ of prop ⊗ prop
  | ∨ of prop ⊗ prop
  | ⊃ of prop ⊗ prop
  | ⊤, ⊥
```

# lcf architecture

```
include LCF
datatype prop =
  | ∧ of prop ⊗ prop
  | ∨ of prop ⊗ prop
  | ⊃ of prop ⊗ prop
  | ⊤, ⊥

datatype jdg = ⊢ of prop dict ⊗ prop
type rule = (jdg, jdg) tactic
```

# lcf architecture

```
include LCF
datatype prop =
  | ∧ of prop ⊗ prop
  | ∨ of prop ⊗ prop
  | ⊃ of prop ⊗ prop
  | ⊤, ⊥

datatype jdg = ⊢ of prop dict ⊗ prop
type rule = (jdg, jdg) tactic

val ∧R, ∨R, ⊃R, ⊤R, ⊥R : rule
val hyp : string → rule
val ∧L : string ⊗ string ⊗ string → rule
...
```

# programs as evidence

**Abstract** **(!!)** type of evidence implemented as functional programming language:

$$
\begin{aligned}
&\textbf{datatype}\ \mathtt{evd} = \\
&\quad\mid\ \mathtt{var}\ \textbf{of}\ \textbf{string} \\
&\quad\mid\ \lambda\ \textbf{of}\ \textbf{string} \otimes \mathtt{evd} \\
&\quad\mid\ \mathtt{ap}\ \textbf{of}\ \mathtt{evd} \otimes \mathtt{evd} \\
&\quad\mid\ \mathtt{pair}\ \textbf{of}\ \mathtt{evd} \otimes \mathtt{evd} \\
&\quad\mid\ \pi_1, \pi_2\ \textbf{of}\ \mathtt{evd} \\
&\quad\mid\ \mathtt{inl}, \mathtt{inr}\ \textbf{of}\ \mathtt{evd} \\
&\quad\mid\ \mathtt{split}\ \textbf{of}\ \mathtt{evd} \otimes \mathtt{evd} \otimes \mathtt{evd}
\end{aligned}
$$

# programs as evidence

**Abstract** **(!!)** type of evidence implemented as functional programming language:

$$
\textbf{datatype}\ \mathsf{evd} =
\begin{array}{|l}
\mathsf{var}\ \textbf{of}\ \textbf{string} \\
\lambda\ \textbf{of}\ \textbf{string} \otimes \mathsf{evd} \\
\mathsf{ap}\ \textbf{of}\ \mathsf{evd} \otimes \mathsf{evd} \\
\mathsf{pair}\ \textbf{of}\ \mathsf{evd} \otimes \mathsf{evd} \\
\pi_1, \pi_2\ \textbf{of}\ \mathsf{evd} \\
\mathsf{inl}, \mathsf{inr}\ \textbf{of}\ \mathsf{evd} \\
\mathsf{split}\ \textbf{of}\ \mathsf{evd} \otimes \mathsf{evd} \otimes \mathsf{evd}
\end{array}
$$

Other options possible: machine code, JavaScript, Perl, PHP, Julia ;-), etc.

# inference rules

inference rule $\Leftrightarrow$ ML function

# inference rules

inference rule $\Leftrightarrow$ ML function

$$\frac{}{\Delta, x : P, \Xi \vdash P}\ hyp[x] \quad \Leftrightarrow \quad \begin{array}{l} \textbf{fun}\ \mathsf{hyp}\,(x)\ \big(\Gamma \vdash P\big) = \\ \quad \textbf{let}\quad (\Delta, Q, \Xi) = \mathsf{split}(\Gamma, x) \\ \quad \textbf{and}\quad \mathsf{true} = (P = Q) \\ \quad \textbf{in}\quad ([\,], \textbf{fn}\ [\,] \Rightarrow \mathsf{var}(x)) \end{array}$$

# inference rules

inference rule $\Leftrightarrow$ ML function

$$\frac{}{\Delta, x : P, \Xi \vdash P}\ hyp[x] \quad \Leftrightarrow \quad \begin{array}{l} \textbf{fun}\ \mathsf{hyp}\,(x)\ \left(\Gamma \vdash P\right) = \\ \quad \textbf{let}\quad (\Delta, Q, \Xi) = \mathsf{split}(\Gamma, x) \\ \quad \textbf{and}\quad \mathsf{true} = (P = Q) \\ \quad \textbf{in}\quad ([\,], \textbf{fn}\ [\,] \Rightarrow \mathsf{var}(x)) \end{array}$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}\ \wedge_R \quad \Leftrightarrow \quad \begin{array}{l} \textbf{fun}\ \wedge_R \left(\Gamma \vdash P \wedge Q\right) = \\ \left( \begin{array}{l} \left[\Gamma \vdash P, \Gamma \vdash Q\right], \\ \textbf{fn}\ [M, N] \Rightarrow \mathsf{pair}(M, N) \end{array} \right) \end{array}$$

# inference rules

$$\frac{\Delta, x : P \wedge Q, y : P, z : Q, \Xi \vdash R}{\Delta, x : P \wedge Q, \Xi \vdash R} \wedge_L[x, y, z]$$

$$\Updownarrow$$

$$
\begin{array}{l}
\textbf{fun } \wedge_L (x, y, z) \; \left( \Gamma \vdash R \right) = \\
\quad \textbf{let} \;\; (\Delta, P \wedge Q, \Xi) = \mathtt{split}(\Gamma, x) \\
\quad \textbf{in} \;\; \left( \begin{array}{l} \left[ \Delta, x : P \wedge Q, y : P, z : Q, \Xi \vdash R \right], \\ \textbf{fn } [M] \Rightarrow [\pi_1(\mathtt{var}(x)), \pi_2(\mathtt{var}(x))/y, z]M \end{array} \right)
\end{array}
$$

# proofs and scripts

$$\overline{x : P \wedge Q \vdash P \wedge Q}$$

$$\Updownarrow$$

# proofs and scripts

$$\dfrac{\overline{x : P \wedge Q, y : P, z : Q \vdash P \wedge Q}}{x : P \wedge Q \vdash P \wedge Q} \;\wedge_L\, [x, y, z]$$

$$\Updownarrow$$

$$\wedge_L\,(x, y, z)$$

# proofs and scripts

$$\dfrac{\dfrac{\overline{x : P \wedge Q, y : P, z : Q \vdash P} \qquad \overline{x : P \wedge Q, y : P, z : Q \vdash Q}}{x : P \wedge Q, y : P, z : Q \vdash P \wedge Q} \wedge_R}{x : P \wedge Q \vdash P \wedge Q} \wedge_L [x, y, z]$$

$$\Updownarrow$$

$\wedge_L (x, y, z)$
then $\wedge_R$

# proofs and scripts

$$\dfrac{\dfrac{}{x : P \wedge Q, y : P, z : Q \vdash P}\ hyp[y] \qquad \dfrac{}{x : P \wedge Q, y : P, z : Q \vdash Q}\ hyp[z]}{\dfrac{x : P \wedge Q, y : P, z : Q \vdash P \wedge Q}{x : P \wedge Q \vdash P \wedge Q}\ \wedge_L\ [x, y, z]}\ \wedge_R$$

$$\Updownarrow$$

$\wedge_L\ (x, y, z)$
then $\wedge_R$
thenl $\left[\begin{array}{l} \mathrm{hyp}(y), \\ \mathrm{hyp}(z) \end{array}\right]$

# proofs and scripts

$$\dfrac{\dfrac{}{x : P \wedge Q, y : P, z : Q \vdash P}\, hyp[y] \qquad \dfrac{}{x : P \wedge Q, y : P, z : Q \vdash Q}\, hyp[z]}{\dfrac{x : P \wedge Q, y : P, z : Q \vdash P \wedge Q}{x : P \wedge Q \vdash P \wedge Q}\, \wedge_L\, [x, y, z]}\, \wedge_R$$

$$\Updownarrow$$

$$\begin{array}{l} \wedge_L\ (x, y, z) \\ \texttt{then}\ \ \wedge_R \\ \texttt{thenl}\ \left[\begin{array}{l} \mathsf{hyp}(y), \\ \mathsf{hyp}(z) \end{array}\right] \end{array} \rightsquigarrow \mathsf{pair}(\pi_1(\mathsf{var}(x)), \pi_2(\mathsf{var}(x)))$$

# free program invariants

A proof synthesizes a program ***(stop calling this "extraction"!).*** Depending on the structure of our logic, we can enforce many invariants!

# free program invariants

A proof synthesizes a program ***(stop calling this "extraction"!).*** Depending on the structure of our logic, we can enforce many invariants!

- ⋆ **Structural invariants:** ordered/affine/linear resource usage

# free program invariants

A proof synthesizes a program ***(stop calling this "extraction"!).*** Depending on the structure of our logic, we can enforce many invariants!

- **Structural invariants:** ordered/affine/linear resource usage
- **Behavioral invariants:** termination, productivity, specification satisfaction

# free program invariants

A proof synthesizes a program ***(stop calling this "extraction"!).*** Depending on the structure of our logic, we can enforce many invariants!

- ⋆ **Structural invariants:** ordered/affine/linear resource usage
- ⋆ **Behavioral invariants:** termination, productivity, specification satisfaction
- ⋆ **Cost invariants?**

# free program invariants

A proof synthesizes a program ***(stop calling this "extraction"!).*** Depending on the structure of our logic, we can enforce many invariants!

- ⋆ **Structural invariants:** ordered/affine/linear resource usage
- ⋆ **Behavioral invariants:** termination, productivity, specification satisfaction
- ⋆ **Cost invariants?**

All this is possible, whilst generating efficient codes in an arbitrary language. Proof structure does not appear in programs.

orthodox lcf architecture

# orthodox lcf architecture

- sequent calculus rules trivially translated into ML
- easy to check that a collection of rules is correct
- **THESE RULES ARE** *definitive*

# orthodox lcf architecture

- sequent calculus rules trivially translated into ML
- easy to check that a collection of rules is correct
- **THESE RULES ARE** *definitive*
- data abstraction guarantees provenience of evidence

# orthodox lcf architecture

- sequent calculus rules trivially translated into ML
- easy to check that a collection of rules is correct
- **THESE RULES ARE *definitive***
- data abstraction guarantees provenience of evidence
- “decidable” typechecking ***completely irrelevant***: type membership just another judgment

# orthodox lcf architecture

- ⋆ sequent calculus rules trivially translated into ML
- ⋆ easy to check that a collection of rules is correct
- ⋆ **THESE RULES ARE *definitive***
- ⋆ data abstraction guarantees provenience of evidence
- ⋆ “decidable” typechecking ***completely irrelevant***: type membership just another judgment
- ⋆ “independently checkable evidence” likewise a distraction[1], because of the soundness of the rules

[1] full employment for purveyors of proof assistants

# orthodox lcf architecture

- ⋆ sequent calculus rules trivially translated into ML
- ⋆ easy to check that a collection of rules is correct
- ⋆ **THESE RULES ARE *definitive***
- ⋆ data abstraction guarantees provenience of evidence
- ⋆ “decidable” typechecking ***completely irrelevant***: type membership just another judgment
- ⋆ “independently checkable evidence” likewise a distraction[1], because of the soundness of the rules

**DECISIVELY SMASH THE FORMALIST CLIQUE!**

[1] full employment for purveyors of proof assistants

# orthodox lcf architecture

there were some problems...

# orthodox lcf architecture

**there were some problems...**

- ⋆ sadly, no dependent refinement (cf. “constructible subgoals property”)

# orthodox lcf architecture

**there were some problems...**

- sadly, no dependent refinement (cf. “constructible subgoals property”)
- no existential variables and unification in core LCF framework (compromises soundness for some logics)

# orthodox lcf architecture

**there were some problems...**

- sadly, no dependent refinement (cf. “constructible subgoals property”)
- no existential variables and unification in core LCF framework (compromises soundness for some logics)
- many sensible rules cannot be encoded (e.g. bidirectional typing)

# orthodox lcf architecture

**there were some problems...**

- ⋆ sadly, no dependent refinement (cf. “constructible subgoals property”)
- ⋆ no existential variables and unification in core LCF framework (compromises soundness for some logics)
- ⋆ many sensible rules cannot be encoded (e.g. bidirectional typing)
- ⋆ complicated and brittle tactics are necessary for basic use

# outline

1. The LCF/PRL Tradition

2. **The Revisionists And Their Running Dogs**

3. RedPRL: New Synthesis Of Proof Refinement

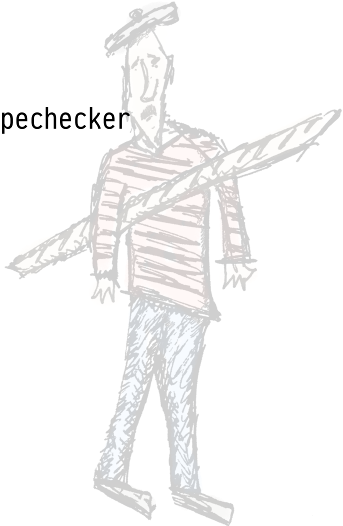
# *revisionist* coq architecture

# *revisionist* coq architecture

- untrusted, non-definitive rules

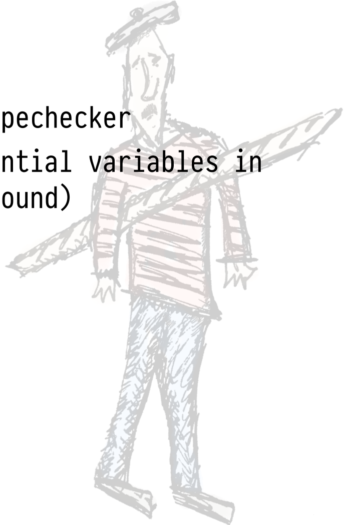
# *revisionist* coq architecture

- ⋆ untrusted, non-definitive rules
- ⋆ core language with definitive typechecker



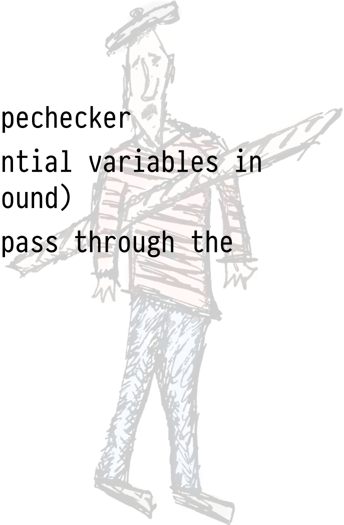
# *revisionist* coq architecture

- untrusted, non-definitive rules
- core language with definitive typechecker
- non-local unification and existential variables in rules and tactics (generally unsound)

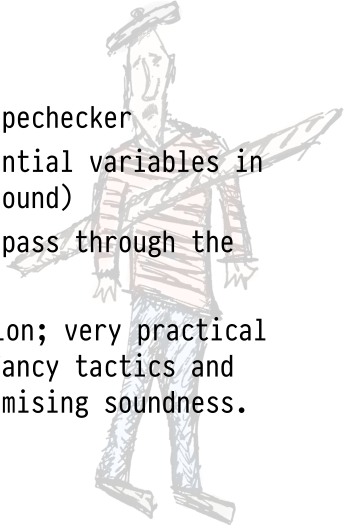


# *revisionist* coq architecture

- untrusted, non-definitive rules
- core language with definitive typechecker
- non-local unification and existential variables in rules and tactics (generally unsound)
- “fine”, because everything must pass through the typechecker



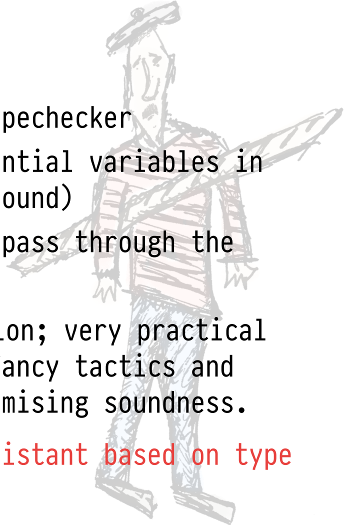
# *revisionist* coq architecture



- ⋆ untrusted, non-definitive rules
- ⋆ core language with definitive typechecker
- ⋆ non-local unification and existential variables in rules and tactics (generally unsound)
- ⋆ “fine”, because everything must pass through the typechecker

**Advantages:** excellent proof automation; very practical in many cases; can experiment with fancy tactics and refinement strategies without compromising soundness.

# *revisionist* coq architecture



- untrusted, non-definitive rules
- core language with definitive typechecker
- non-local unification and existential variables in rules and tactics (generally unsound)
- "fine", because everything must pass through the typechecker

**Advantages:** excellent proof automation; very practical in many cases; can experiment with fancy tactics and refinement strategies without compromising soundness.

Coq is the most successful proof assistant based on type theory in history.

*revisionist* coq architecture

Disadvantages of revisionism

# *revisionist* coq architecture

**Disadvantages of revisionism**

1. dangling existential variables (partly mitigated)

# *revisionist* coq architecture

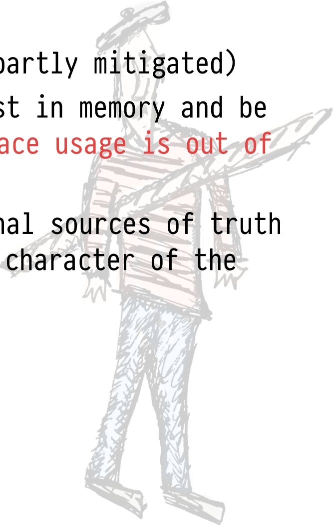
**Disadvantages of revisionism**

1. dangling existential variables (partly mitigated)
2. obscenely large objects must exist in memory and be shoved through a typechecker: space usage is out of control, easy to wedge Coq

# *revisionist* coq architecture

**Disadvantages of revisionism**

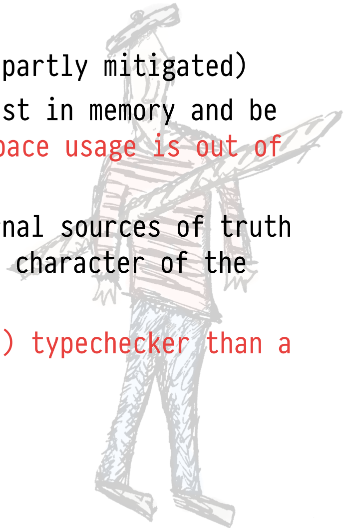
1. dangling existential variables (partly mitigated)
2. obscenely large objects must exist in memory and be shoved through a typechecker: space usage is out of control, easy to wedge Coq
3. not clear how to integrate external sources of truth (solvers) without destroying the character of the logic



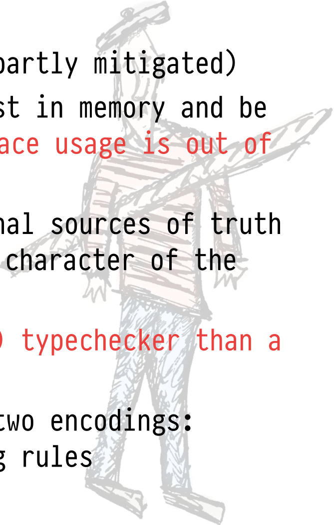
# *revisionist* coq architecture

**Disadvantages of revisionism**

1. dangling existential variables (partly mitigated)
2. obscenely large objects must exist in memory and be shoved through a typechecker: space usage is out of control, easy to wedge Coq
3. not clear how to integrate external sources of truth (solvers) without destroying the character of the logic
4. more difficult to verify a (real) typechecker than a (real) refiner



# *revisionist* coq architecture



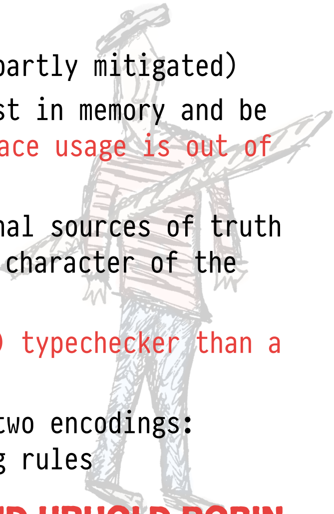
**Disadvantages of revisionism**

1. dangling existential variables (partly mitigated)
2. obscenely large objects must exist in memory and be shoved through a typechecker: space usage is out of control, easy to wedge Coq
3. not clear how to integrate external sources of truth (solvers) without destroying the character of the logic
4. more difficult to verify a (real) typechecker than a (real) refiner
5. logical rules are duplicated in two encodings: refinement rules and typechecking rules

# *revisionist* coq architecture

**Disadvantages of revisionism**

1. dangling existential variables (partly mitigated)
2. obscenely large objects must exist in memory and be shoved through a typechecker: space usage is out of control, easy to wedge Coq
3. not clear how to integrate external sources of truth (solvers) without destroying the character of the logic
4. more difficult to verify a (real) typechecker than a (real) refiner
5. logical rules are duplicated in two encodings: refinement rules and typechecking rules



**RESIST FRENCH IMPERIALISM AND UPHOLD ROBIN MILNER THOUGHT!**

# outline

1. The LCF/PRL Tradition
2. The Revisionists And Their Running Dogs
3. RedPRL: New Synthesis Of Proof Refinement

# RedPRL: dependent refinement

RedPRL is a return to **orthodoxy**, synthesizing modern developments in proof refinement.

# RedPRL: dependent refinement

RedPRL is a return to **orthodoxy**, synthesizing modern developments in proof refinement.

- ⋆ ***Dependent LCF*:** each subgoal induces a metavariable that can be used in the statements of later subgoals.

# RedPRL: dependent refinement

RedPRL is a return to **orthodoxy**, synthesizing modern developments in proof refinement.

- ***Dependent LCF***: each subgoal induces a metavariable that can be used in the statements of later subgoals.
- **Metavariables can only be resolved *locally***, by refinement rules (NOT UNIFICATION).

# RedPRL: dependent refinement

RedPRL is a return to **orthodoxy**, synthesizing modern developments in proof refinement.

- ***Dependent LCF***: each subgoal induces a metavariable that can be used in the statements of later subgoals.
- **Metavariables can only be resolved** ***locally***, by refinement rules (NOT UNIFICATION).
- Strict $1 \leftrightarrow 1$ relation between goals and metavariables in context.

# RedPRL: dependent refinement

RedPRL is a return to **orthodoxy**, synthesizing modern developments in proof refinement.

- ⋆ ***Dependent LCF***: each subgoal induces a metavariable that can be used in the statements of later subgoals.
- ⋆ **Metavariables can only be resolved *locally***, by refinement rules (NOT UNIFICATION).
- ⋆ Strict $1 \leftrightarrow 1$ relation between goals and metavariables in context.
- ⋆ Precisely what is needed to encode bidirectional typing rules.

# remark on unification

Resolving existential variables via unification is so much fun! **But it induces non-local soundness conditions for a refiner** (very sad!).

# remark on unification

Resolving existential variables via unification is so much fun! **But it induces non-local soundness conditions for a refiner** (very sad!).

1. Changes the character of the implemented logic: can make type theory anti-classical if not careful (cf. Agda).

# remark on unification

Resolving existential variables via unification is so much fun! **But it induces non-local soundness conditions for a refiner** (very sad!).

1. Changes the character of the implemented logic: can make type theory anti-classical if not careful (cf. Agda).
2. Probably unsound in the presence of subtyping and non-discrete equality (e.g. Nuprl).

# remark on unification

Resolving existential variables via unification is so much fun! **But it induces non-local soundness conditions for a refiner** (very sad!).

1. Changes the character of the implemented logic: can make type theory anti-classical if not careful (cf. Agda).
2. Probably unsound in the presence of subtyping and non-discrete equality (e.g. Nuprl).

⋆ Works out fine in Coq because the refinement rules do not need to be sound.

# remark on unification

Resolving existential variables via unification is so much fun! **But it induces non-local soundness conditions for a refiner** (very sad!).

1. Changes the character of the implemented logic: can make type theory anti-classical if not careful (cf. Agda).
2. Probably unsound in the presence of subtyping and non-discrete equality (e.g. Nuprl).

- ⋆ Works out fine in Coq because the refinement rules do not need to be sound.
- ⋆ Unification must be integrated as a judgment in your theory, not as part of a refinement framework. See Cockx/Devriese/Piessens ICFP 2016.

# dependent lcf: example

Enables a straightforward encoding of sophisticated dependent refinement rules which are not expressible in Classic LCF or Nuprl.

# *dependent lcf*: example

Enables a straightforward encoding of sophisticated dependent refinement rules which are not expressible in **Classic LCF** or Nuprl. For example...

$$\frac{\Gamma \vdash R \Rightarrow (x : A) \to B_x \quad \Gamma \vdash S \Leftarrow A}{\Gamma \vdash R(S) \Rightarrow [S/x]B_x} \ \text{synth/ap}$$

# *dependent lcf*: example

Enables a straightforward encoding of sophisticated dependent refinement rules which are not expressible in **Classic LCF** or Nuprl. For example...

$$\frac{\Gamma \vdash R \Rightarrow (x : A) \to B_x \quad \Gamma \vdash S \Leftarrow A}{\Gamma \vdash R(S) \Rightarrow [S/x]B_x} \text{ synth/ap}$$

$$\frac{\Gamma \vdash \overset{(M)}{A} \ \textit{true} \quad \Gamma \vdash [M/x]B_x \ \textit{true}}{\Gamma \vdash (x : A) \times B_x \ \textit{true}} \text{ intro/}\Sigma$$

# use *dependent lcf* today!

Implemented as a modular **Standard ML** library, which you can use in your own project today!

https://github.com/RedPRL/sml-dependent-lcf

# use *dependent lcf* today!

Implemented as a modular **Standard ML** library, which you can use in your own project today!

https://github.com/RedPRL/sml-dependent-lcf

Restricts automatically to **Classic LCF** when instantiated without dependency/substitution structure.

# RedPRL's cubical type theory

$$
\begin{tikzcd}
\sqcup^n_k \arrow[r] \arrow[d, hook] \arrow[dr, phantom, "\star"] & \chi \\
\square^n \arrow[ur, dashed] &
\end{tikzcd}
$$

# RedPRL's cubical type theory

**Computational Higher-Dimensional Type Theory**
[Angiuli/Harper/Wilson POPL 2017]

# RedPRL's cubical type theory

**Computational Higher-Dimensional Type Theory**
[Angiuli/Harper/Wilson POPL 2017]

* a type theory with both extensional ***equality*** and intensional ***identification*** (paths)

# RedPRL's cubical type theory

**Computational Higher-Dimensional Type Theory**
[Angiuli/Harper/Wilson POPL 2017]

- a type theory with both extensional ***equality*** and intensional ***identification*** (paths)
- higher inductive types: the circle

# RedPRL's cubical type theory

**Computational Higher-Dimensional Type Theory**
[Angiuli/Harper/Wilson POPL 2017]

- ⋆ a type theory with both extensional ***equality*** and intensional ***identification*** (paths)
- ⋆ higher inductive types: the circle
- ⋆ strict types: strict booleans (new)

# RedPRL's cubical type theory

**Computational Higher-Dimensional Type Theory**
[Angiuli/Harper/Wilson POPL 2017]

- ⋆ a type theory with both extensional ***equality*** and intensional ***identification*** (paths)
- ⋆ higher inductive types: the circle
- ⋆ strict types: strict booleans (new)
- ⋆ computational canonicity (previous Licata/Harper result established canonicity up to ***judgmental equality*** for 2D type theory)

# RedPRL's cubical type theory

**Computational Higher-Dimensional Type Theory**
[Angiuli/Harper/Wilson POPL 2017]

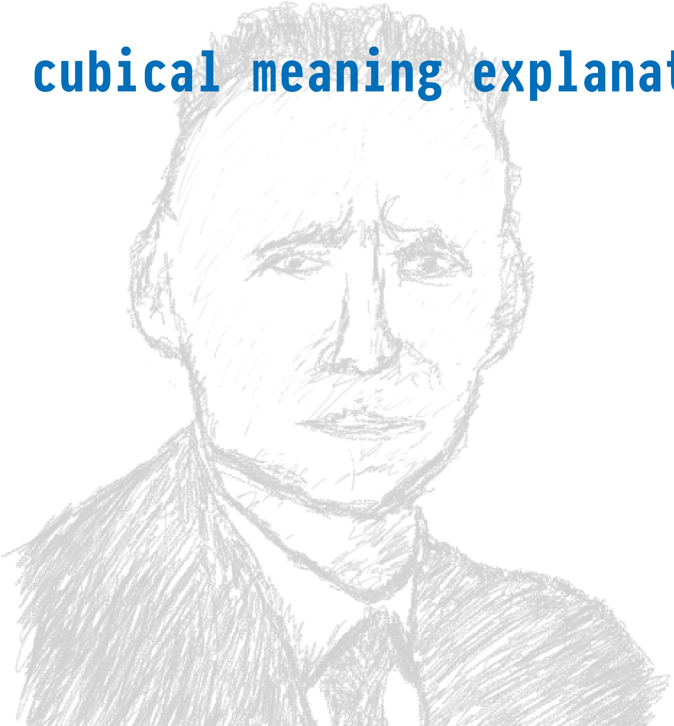
- a type theory with both extensional ***equality*** and intensional ***identification*** (paths)
- higher inductive types: the circle
- strict types: strict booleans (new)
- computational canonicity (previous Licata/Harper result established canonicity up to ***judgmental equality*** for 2D type theory)
- an instance of univalence, $\mathsf{not}_x$.

# RedPRL's cubical type theory

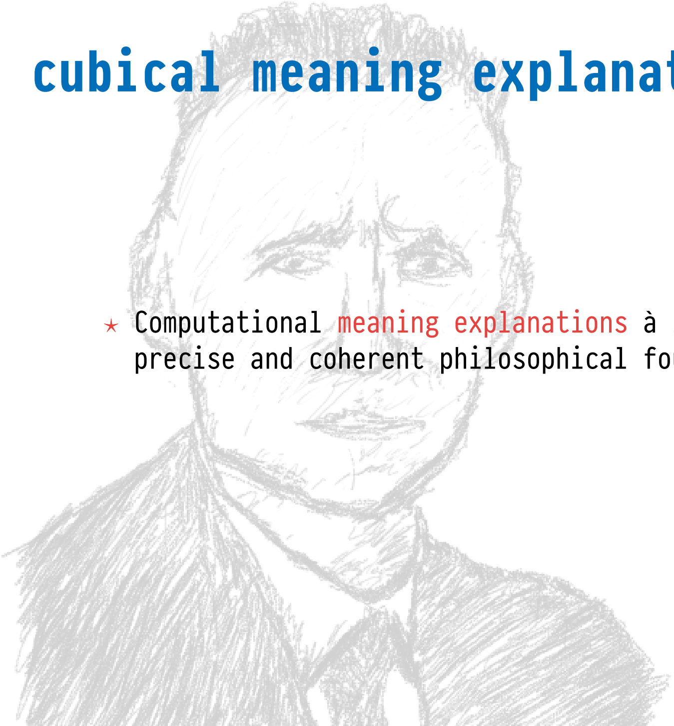
**Computational Higher-Dimensional Type Theory**
[Angiuli/Harper/Wilson POPL 2017]

- ⋆ a type theory with both extensional ***equality*** and intensional ***identification*** (paths)
- ⋆ higher inductive types: the circle
- ⋆ strict types: strict booleans (new)
- ⋆ computational canonicity (previous Licata/Harper result established canonicity up to ***judgmental equality*** for 2D type theory)
- ⋆ an instance of univalence, $\mathrm{not}_x$.
- ⋆ a deterministic and type-free operational semantics, amenable to cost analysis.

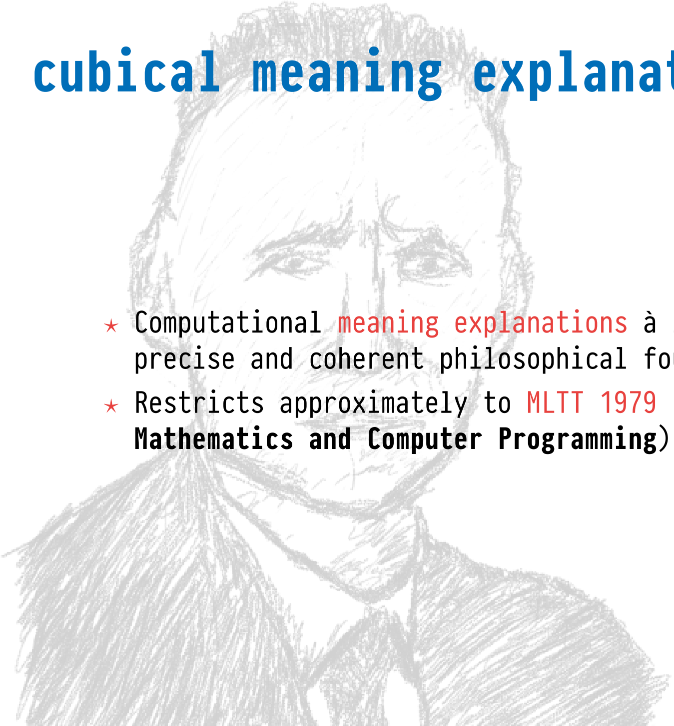
# cubical meaning explanation



# cubical meaning explanation



- ⋆ Computational meaning explanations à la Martin-Löf: precise and coherent philosophical foundation.



# cubical meaning explanation

- ⋆ Computational meaning explanations à la Martin-Löf: precise and coherent philosophical foundation.
- ⋆ Restricts approximately to MLTT 1979 (**Constructive Mathematics and Computer Programming**) at dimension 0.


COQ