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.

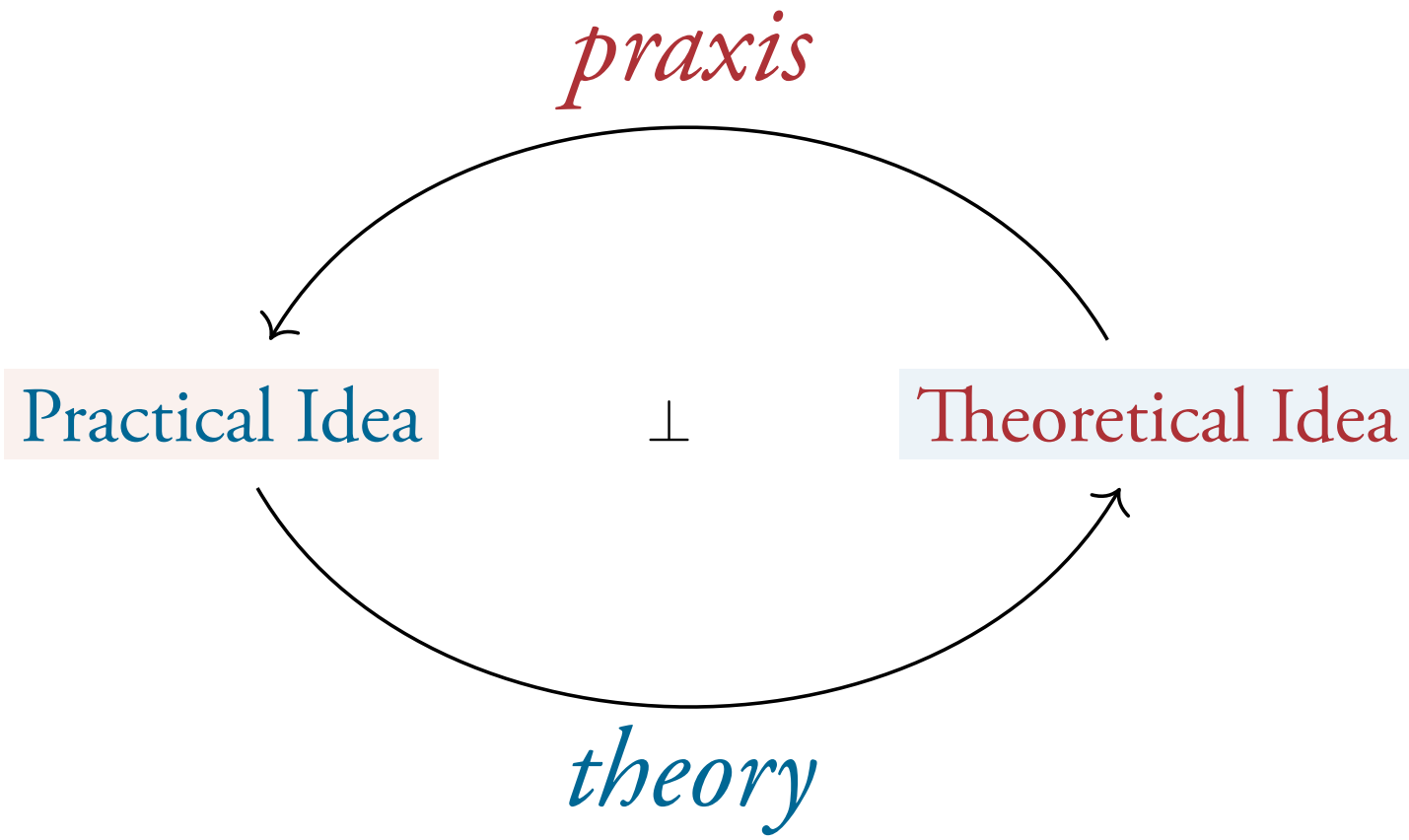
what is RedPRL?

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.

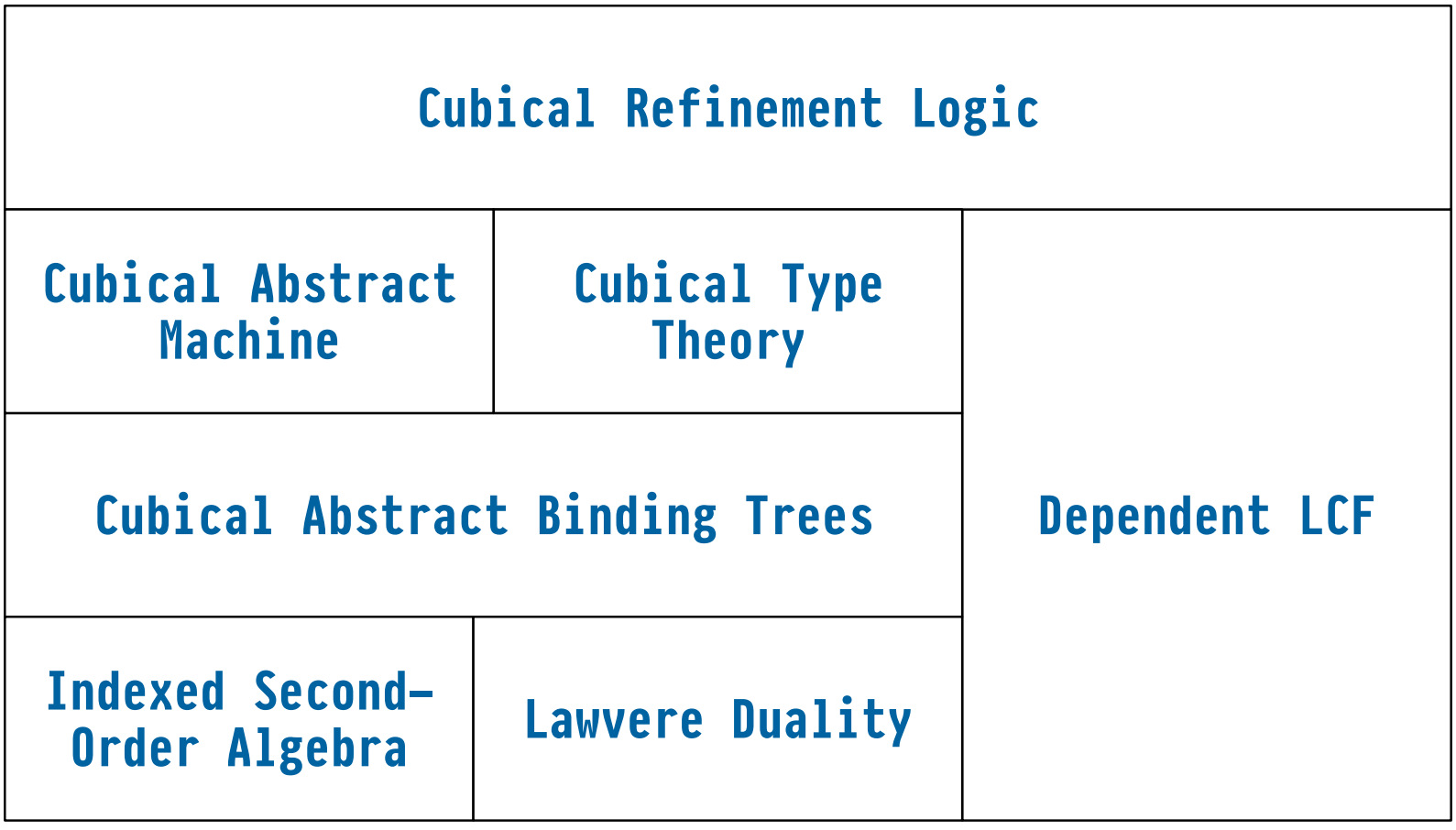
So, why bother?

the absolute idea



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

overview of RedPRL



outline

1. The LCF 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:

```
datatype evd =
  | var of string
  | λ of string ⊗ evd
  | ap of evd ⊗ evd
  | pair of evd ⊗ evd
  | π1, π2 of evd
  | inl, inr of evd
  | split of evd ⊗ evd ⊗ evd
```

programs as evidence

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

```
datatype evd =
    var of string
  | λ of string ⊗ evd
  | ap of evd ⊗ evd
  | pair of evd ⊗ evd
  | π1, π2 of evd
  | inl, inr of evd
  | split of evd ⊗ evd ⊗ evd
```

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

inference rules

`inference rule` $\Leftrightarrow$ `ML function`

inference rules

inference rule $\Leftrightarrow$ ML function

$$\frac{}{\Delta, x : P, \Xi \vdash P}\ hyp[x] \quad \Leftrightarrow$$

```
fun hyp (x) (Γ ⊢ P) =
 let  (Δ, Q, Ξ) = split(Γ, x)
 and  true = (P = Q)
  in  ([], fn [] ⇒ var(x))
```

inference rules

$$\texttt{inference rule} \Leftrightarrow \texttt{ML function}$$

$$\frac{}{\Delta, x : P, \Xi \vdash P}\ hyp[x] \quad \Leftrightarrow$$

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

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}\ \wedge_R \quad \Leftrightarrow$$

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

inference rules

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

$$\Updownarrow$$

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

proofs and scripts

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

proofs and scripts

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

$$\Updownarrow$$

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

proofs and scripts

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

$$\Updownarrow$$

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

proofs and scripts

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

$$\Updownarrow$$

$$\begin{array}{l} \wedge_L (x, y, z) \\ \texttt{then}\ \ \wedge_R \\ \texttt{then1}\ \left[\begin{array}{l} \texttt{hyp}(y), \\ \texttt{hyp}(z) \end{array}\right] \end{array}$$

proofs and scripts

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

$\Updownarrow$

$$\wedge_L\,(x, y, z)\ \texttt{then}\ \wedge_R\ \texttt{thenl}\ \begin{bmatrix} \texttt{hyp}(y), \\ \texttt{hyp}(z) \end{bmatrix} \rightsquigarrow \texttt{pair}(\pi_1(\texttt{var}(x)), \pi_2(\texttt{var}(x)))$$

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 need to 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
- "independently checkable evidence" a ***complete 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
- "independently checkable evidence" a *complete distraction*[1], because of the soundness of the rules
- type membership just another judgment, no need for it to be decidable

[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
- "independently checkable evidence" a ***complete distraction***[1], because of the soundness of the rules
- type membership just another judgment, no need for it to be decidable

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”)
- ⋆ ⇒ many sensible rules cannot be encoded (e.g. intro rule for Σ-types, bidirectional typing)

orthodox lcf architecture

there were some problems...

- ⋆ sadly, no dependent refinement (cf. “constructible subgoals property”)
- ⋆ ⇒ many sensible rules cannot be encoded (e.g. intro rule for Σ-types, bidirectional typing)
- ⋆ ⇒ complicated and brittle tactics are necessary for basic use

outline

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

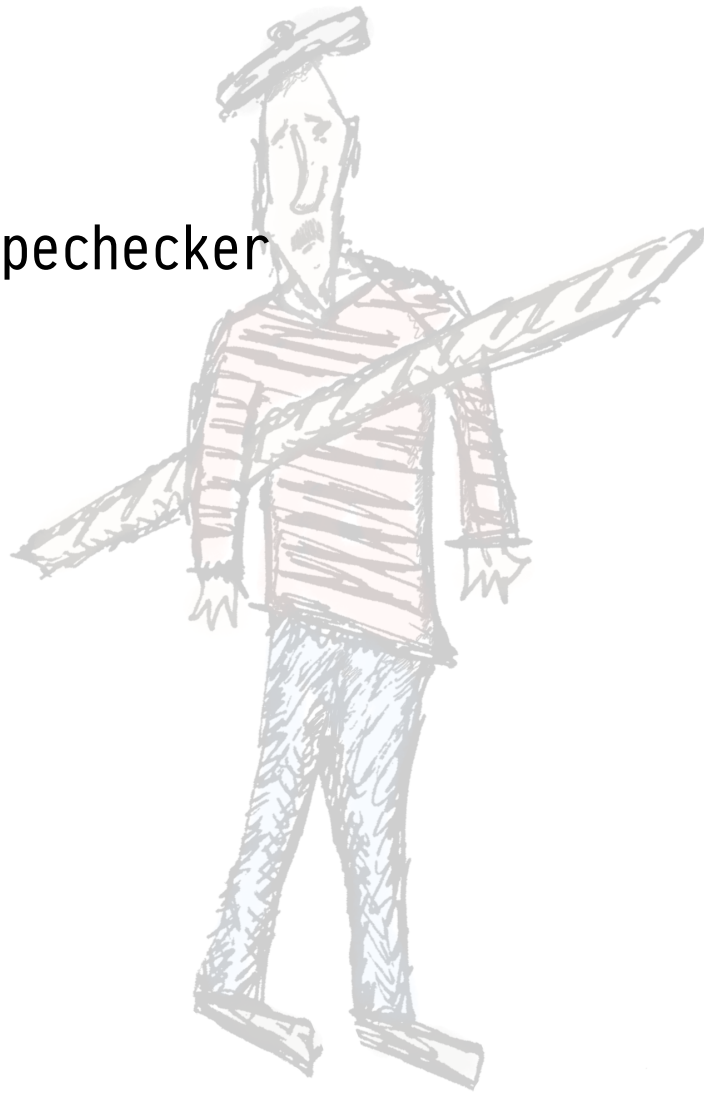
revisionist coq architecture

revisionist coq architecture

- untrusted, non-definitive rules

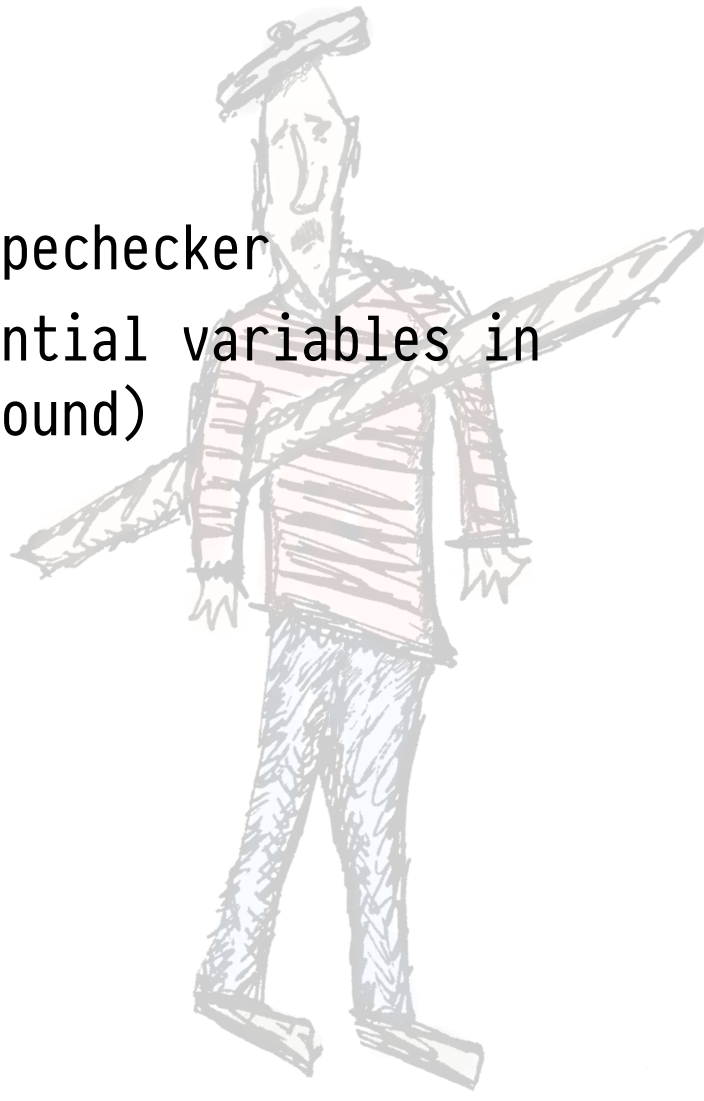
revisionist coq architecture

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



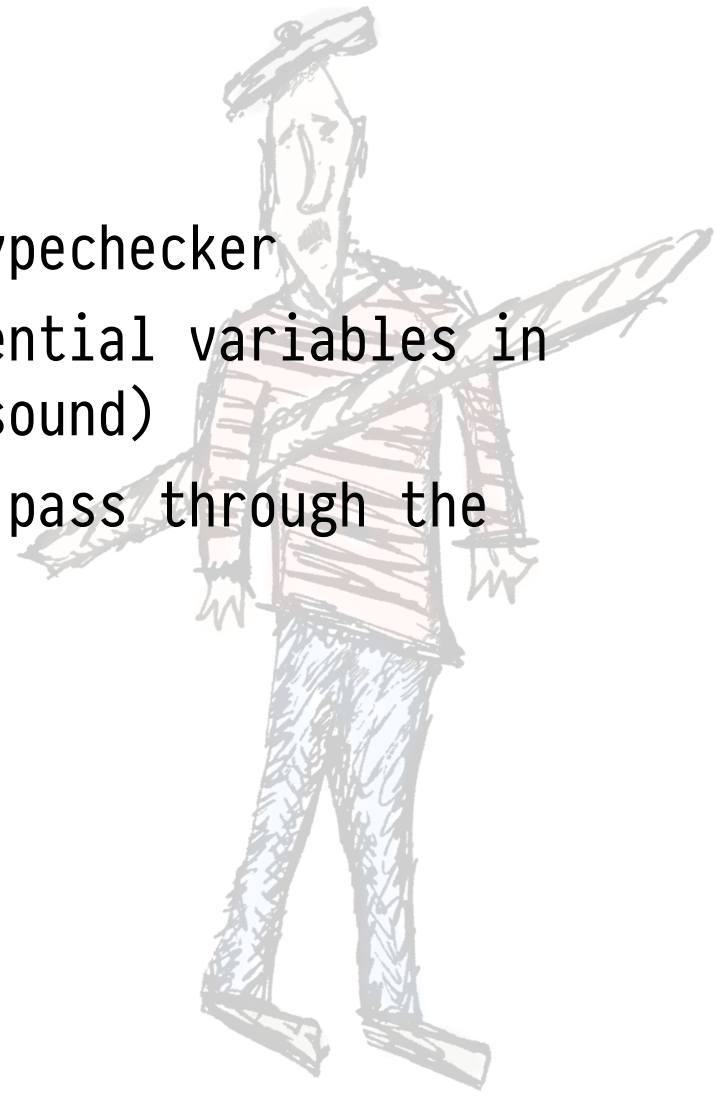
revisionist coq architecture

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



revisionist coq architecture

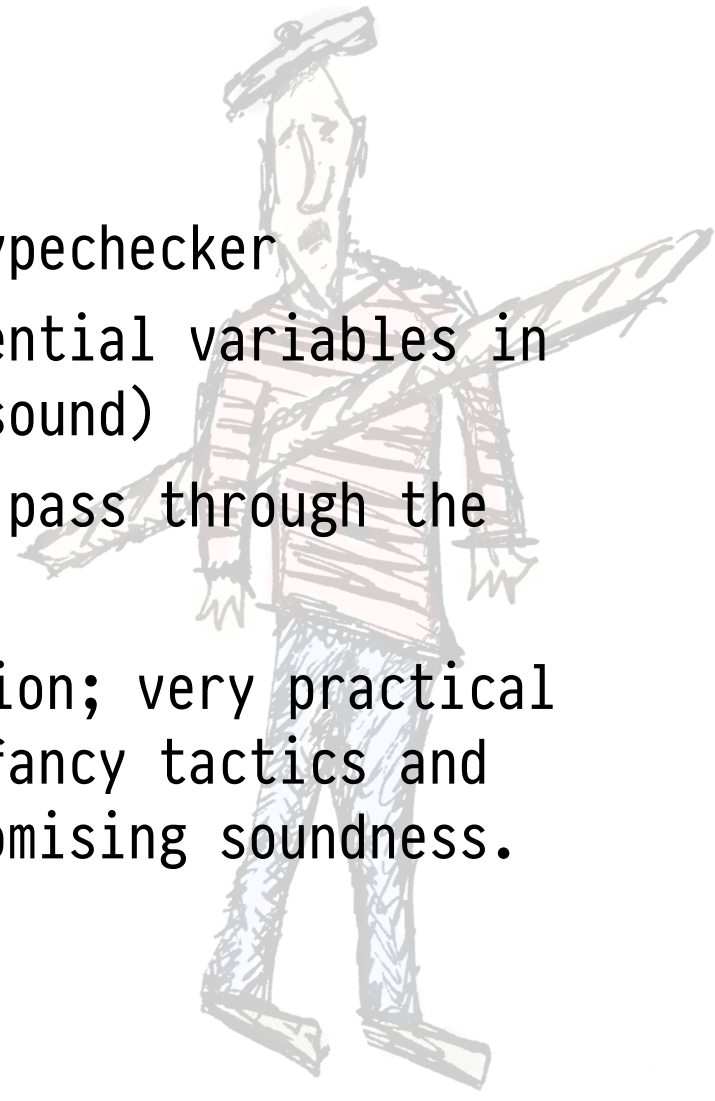
- untrusted, non-definitive rules
- core language with definitive typechecker
- non-local unification and existential variables in rules and tactics (generally unsound)
- “fine”, because everything must pass through the typechecker



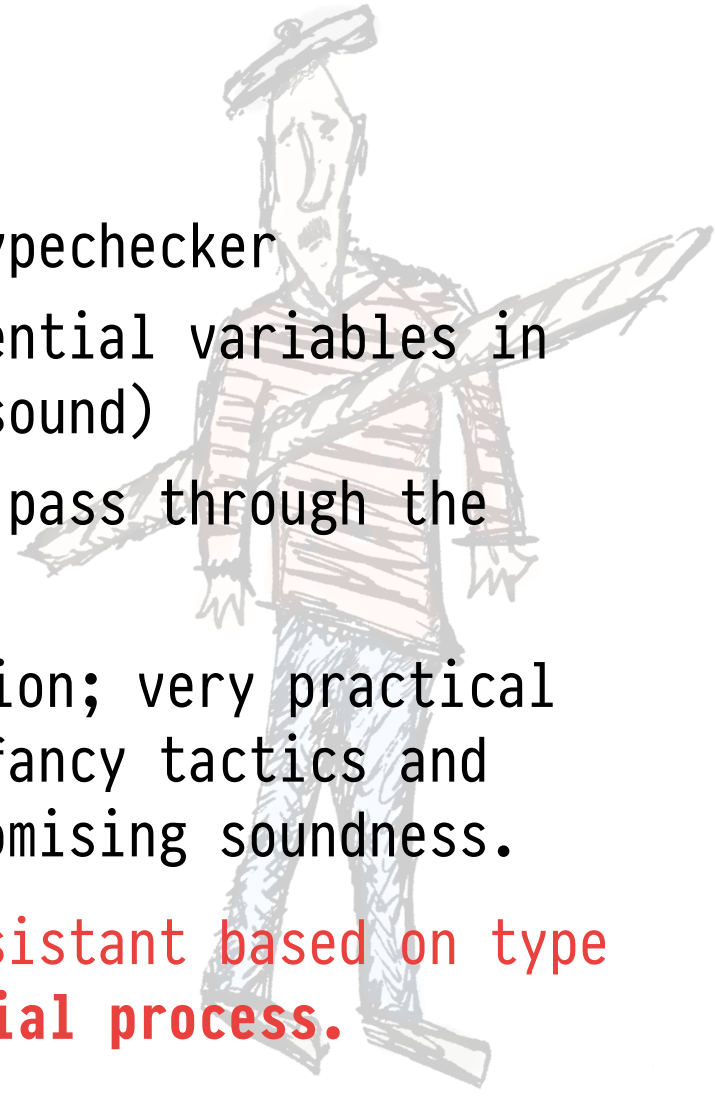
revisionist coq architecture

- 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. **Science is a social process.**

revisionist coq architecture

Disadvantages of revisionism

revisionist coq architecture

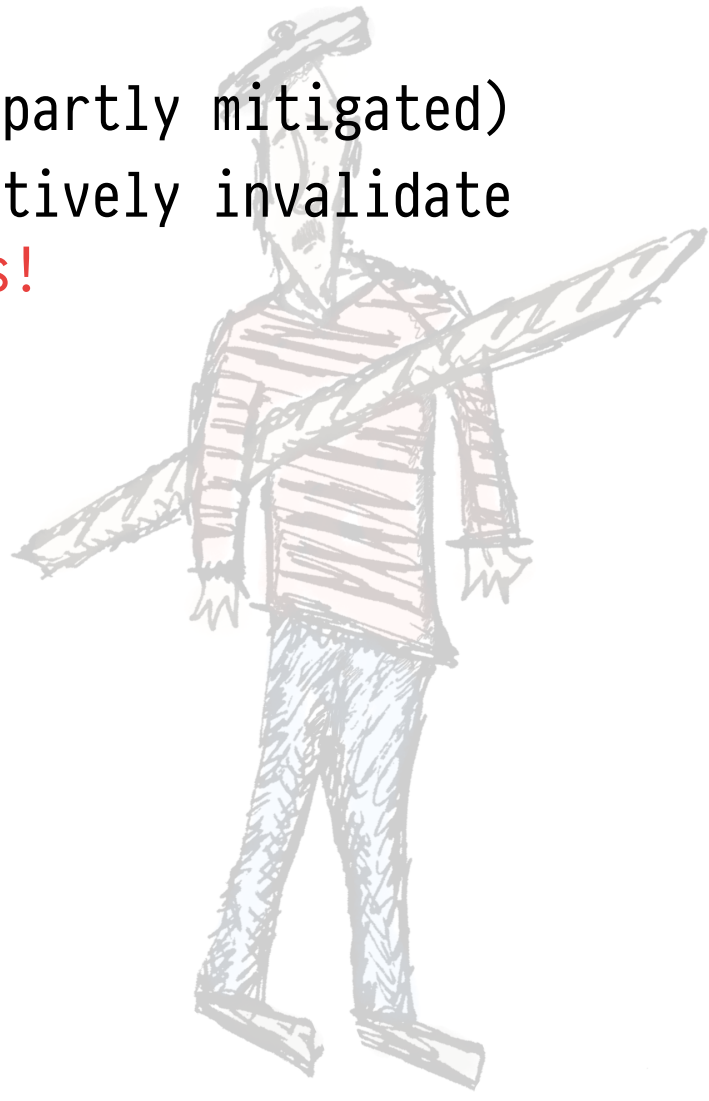
Disadvantages of revisionism

1. dangling existential variables (partly mitigated)

revisionist coq architecture

Disadvantages of revisionism

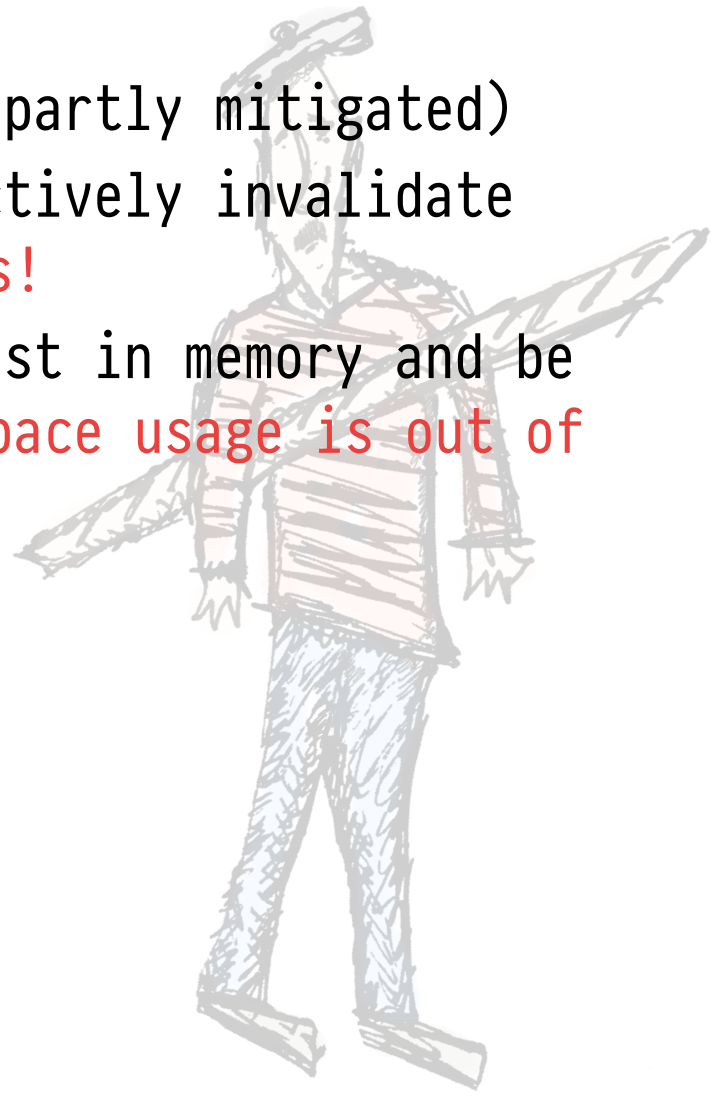
1. dangling existential variables (partly mitigated)
2. proof of one subgoal can retroactively invalidate another completed subgoal! yikes!



revisionist coq architecture

Disadvantages of revisionism

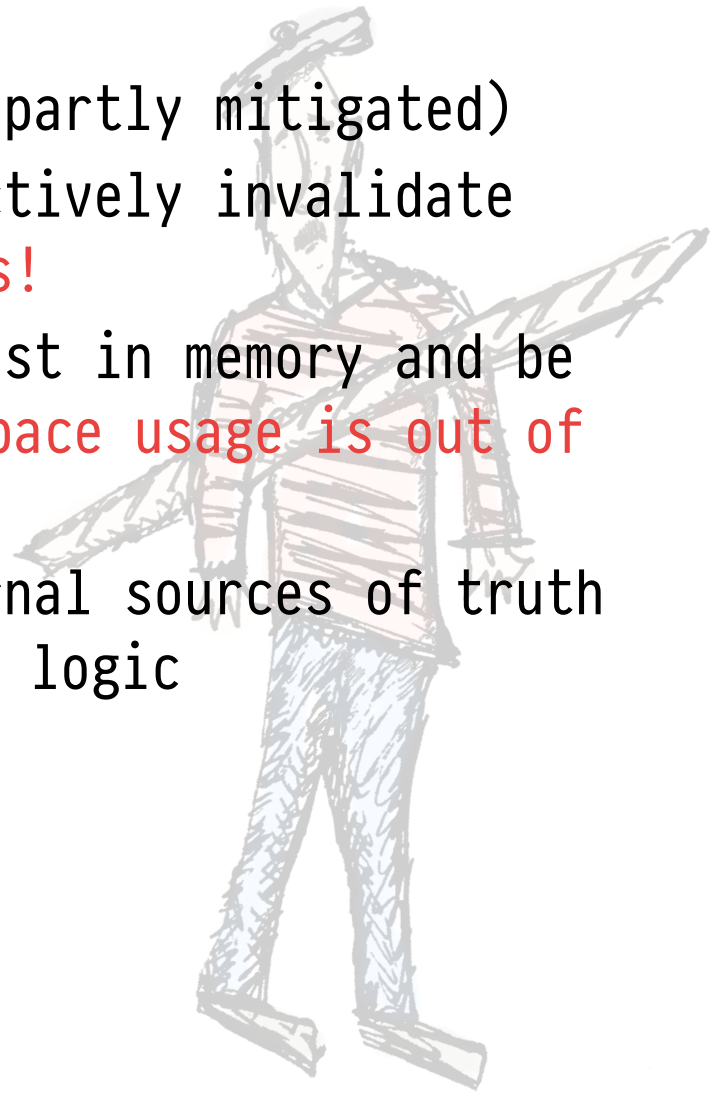
1. dangling existential variables (partly mitigated)
2. proof of one subgoal can retroactively invalidate another completed subgoal! yikes!
3. obscenely large objects must exist in memory and be shoved through a typechecker: space usage is out of control, easy to wedge Coq



revisionist coq architecture

Disadvantages of revisionism

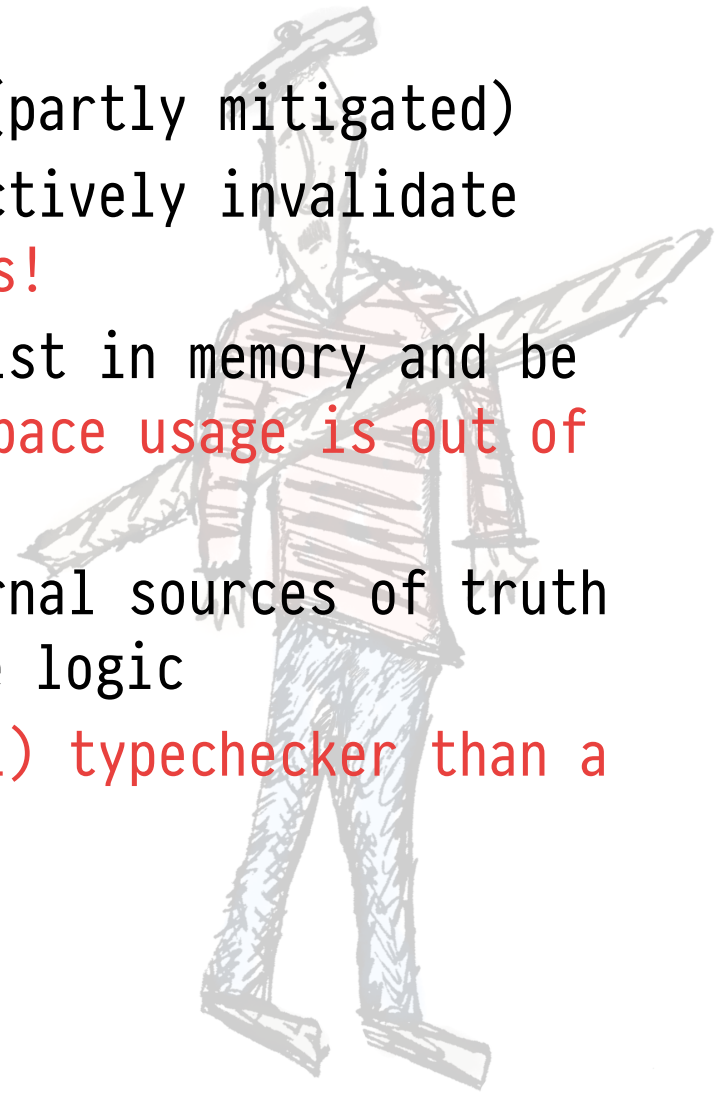
1. dangling existential variables (partly mitigated)
2. proof of one subgoal can retroactively invalidate another completed subgoal! yikes!
3. obscenely large objects must exist in memory and be shoved through a typechecker: space usage is out of control, easy to wedge Coq
4. not clear how to integrate external sources of truth (solvers) without disturbing the logic



revisionist coq architecture

Disadvantages of revisionism

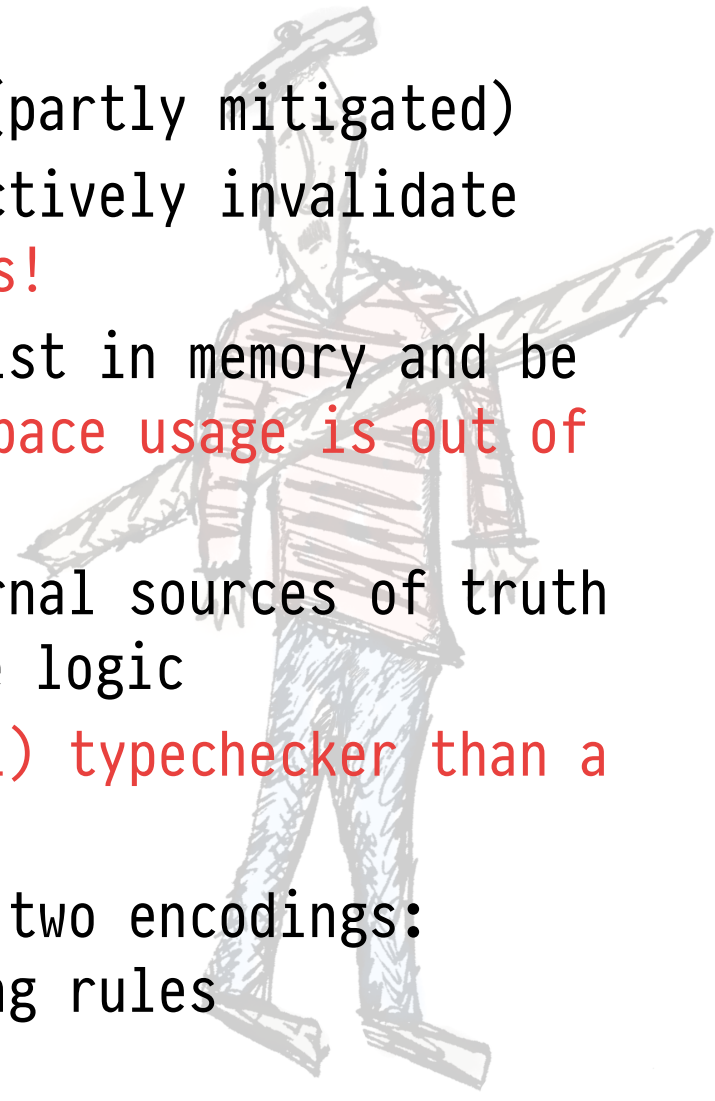
1. dangling existential variables (partly mitigated)
2. proof of one subgoal can retroactively invalidate another completed subgoal! yikes!
3. obscenely large objects must exist in memory and be shoved through a typechecker: space usage is out of control, easy to wedge Coq
4. not clear how to integrate external sources of truth (solvers) without disturbing the logic
5. more difficult to verify a (real) typechecker than a (real) refiner



revisionist coq architecture

Disadvantages of revisionism

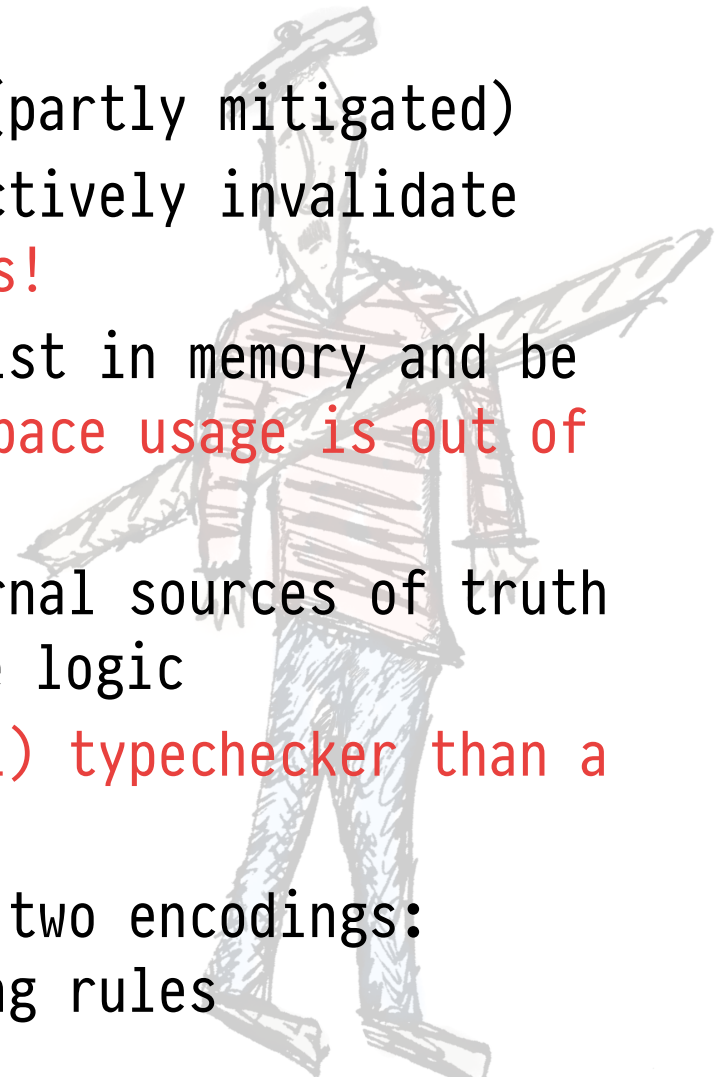
1. dangling existential variables (partly mitigated)
2. proof of one subgoal can retroactively invalidate another completed subgoal! yikes!
3. obscenely large objects must exist in memory and be shoved through a typechecker: space usage is out of control, easy to wedge Coq
4. not clear how to integrate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



RESIST FRENCH IMPERIALISM AND UPHOLD ROBIN MILNER THOUGHT!

outline

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


COO

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).
- Adds nothing to the object logic: just a means of incremental construction / 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.
- **Metavariables can only be resolved *locally***, by refinement rules (NOT UNIFICATION).
- Adds nothing to the object logic: just a means of incremental construction / refinement.
- Precisely what is needed to encode existential instantiation, 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.

from classic lcf to *dependent lcf*

$$\frac{\begin{array}{l}\mathcal{J}_0 \rightsquigarrow \mathfrak{X}_0 \\ \mathcal{J}_1 \rightsquigarrow \mathfrak{X}_1 \\ \vdots \\ \mathcal{J}_n \rightsquigarrow \mathfrak{X}_n\end{array}}{\mathcal{J} \rightsquigarrow [\mathfrak{X}_0, \ldots, \mathfrak{X}_n].M}\ \texttt{my-rule}$$

from classic lcf to *dependent lcf*

$$\frac{\begin{array}{l}[\Omega].\mathcal{J}_0 \rightsquigarrow \mathfrak{X}_0 \\ [\Omega, \mathfrak{X}_0].\mathcal{J}_1 \rightsquigarrow \mathfrak{X}_1 \\ \vdots \\ [\Omega, \mathfrak{X}_0, \ldots, \mathfrak{X}_{n-1}].\mathcal{J}_n \rightsquigarrow \mathfrak{X}_n\end{array}}{[\Omega].\mathcal{J} \rightsquigarrow [\mathfrak{X}_0, \ldots, \mathfrak{X}_n].M}\ \texttt{my-rule}$$

from classic lcf to *dependent lcf*

$$\frac{\begin{array}{l}[\Omega].\mathcal{J}_0 \rightsquigarrow \mathfrak{E}_0 \\ [\Omega, \mathfrak{E}_0].\mathcal{J}_1 \rightsquigarrow \mathfrak{E}_1 \\ \vdots \\ [\Omega, \mathfrak{E}_0, \ldots, \mathfrak{E}_{n-1}].\mathcal{J}_n \rightsquigarrow \mathfrak{E}_n\end{array}}{[\Omega].\mathcal{J} \rightsquigarrow [\mathfrak{E}_0, \ldots, \mathfrak{E}_n].M}\ \textsc{my-rule}$$

- ***lax naturality*** ensures that rules commute with substitution up to approximation

from classic lcf to *dependent lcf*

$$\frac{\begin{array}{l}[\Omega].\mathcal{J}_0 \rightsquigarrow \mathfrak{E}_0 \\ [\Omega, \mathfrak{E}_0].\mathcal{J}_1 \rightsquigarrow \mathfrak{E}_1 \\ \vdots \\ [\Omega, \mathfrak{E}_0, \ldots, \mathfrak{E}_{n-1}].\mathcal{J}_n \rightsquigarrow \mathfrak{E}_n\end{array}}{[\Omega].\mathcal{J} \rightsquigarrow [\mathfrak{E}_0, \ldots, \mathfrak{E}_n].M}\ \text{my-rule}$$

- ***lax naturality*** ensures that rules commute with substitution up to approximation
- gorgeous denotational semantics

from classic lcf to *dependent lcf*

$$\frac{\begin{array}{l}[\Omega].\mathcal{J}_0 \rightsquigarrow \mathfrak{E}_0 \\ [\Omega, \mathfrak{E}_0].\mathcal{J}_1 \rightsquigarrow \mathfrak{E}_1 \\ \vdots \\ [\Omega, \mathfrak{E}_0, \ldots, \mathfrak{E}_{n-1}].\mathcal{J}_n \rightsquigarrow \mathfrak{E}_n\end{array}}{[\Omega].\mathcal{J} \rightsquigarrow [\mathfrak{E}_0, \ldots, \mathfrak{E}_n].M}\ \text{my-rule}$$

- ***lax naturality*** ensures that rules commute with substitution up to approximation
- gorgeous denotational semantics
- EASY to implement. (maybe not super efficient! refinement machine future work.)

from classic lcf to *dependent lcf*

$$\frac{\begin{array}{l}[\Omega].\mathcal{J}_0 \rightsquigarrow \mathfrak{E}_0 \\ [\Omega, \mathfrak{E}_0].\mathcal{J}_1 \rightsquigarrow \mathfrak{E}_1 \\ \vdots \\ [\Omega, \mathfrak{E}_0, \ldots, \mathfrak{E}_{n-1}].\mathcal{J}_n \rightsquigarrow \mathfrak{E}_n\end{array}}{[\Omega].\mathcal{J} \rightsquigarrow [\mathfrak{E}_0, \ldots, \mathfrak{E}_n].M}\ \text{my-rule}$$

- ⋆ ***lax naturality*** ensures that rules commute with substitution up to approximation
- ⋆ gorgeous denotational semantics
- ⋆ EASY to implement. (maybe not super efficient! refinement machine future work.)
- ⋆ dependent refinement = maximum parallelism of proof acts

dependent Lcf: examples

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

dependent lcf: examples

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

$$\frac{\begin{array}{l}[\Omega].\,\Gamma \vdash A\ \mathit{true} \rightsquigarrow \mathtt{m} \\ [\Omega, \mathtt{m}].\,\Gamma \vdash B[\mathtt{m}]\ \mathit{true} \rightsquigarrow \mathtt{n}\end{array}}{[\Omega].\,\Gamma \vdash (x : A) \times B[x]\ \mathit{true} \rightsquigarrow [\Omega, \mathtt{m}, \mathtt{n}].\,\langle \mathtt{m}, \mathtt{n} \rangle}\ \text{intro}/\Sigma$$

dependent LCF: examples

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

$$\frac{[\Omega].\,\Gamma \vdash A\ \mathit{true} \rightsquigarrow \mathtt{m} \qquad [\Omega, \mathtt{m}].\,\Gamma \vdash B[\mathtt{m}]\ \mathit{true} \rightsquigarrow \mathtt{n}}{[\Omega].\,\Gamma \vdash (x : A) \times B[x]\ \mathit{true} \rightsquigarrow [\Omega, \mathtt{m}, \mathtt{n}].\,\langle \mathtt{m}, \mathtt{n} \rangle}\ \text{intro}/\Sigma$$

wow!!

dependent lcf: examples

A more sophisticated example: bidirectional typing rules (not just for typecheckers!—crucial for automation).

dependent lcf: examples

A more sophisticated example: bidirectional typing rules (not just for typecheckers!—crucial for automation).

$$\frac{\begin{array}{l}[\Omega].\,\Gamma \vdash R\ \mathit{synth} \rightsquigarrow \mathtt{ty} \\ [\Omega, \mathtt{ty}].\,\mathtt{ty}\ \mathit{match}\{0\}\ \mathtt{dfun} \rightsquigarrow \mathtt{a} \\ [\Omega, \mathtt{ty}, \mathtt{a}].\,\mathtt{ty}\ \mathit{match}\{1\}\ \mathtt{dfun} \rightsquigarrow \mathtt{b} \\ [\Omega, \mathtt{ty}, \mathtt{a}, \mathtt{b}].\,S \in \mathtt{a} \rightsquigarrow \_\end{array}}{[\Omega].\,\Gamma \vdash R(S)\ \mathit{synth} \rightsquigarrow [\Omega, \mathtt{ty}, \mathtt{a}, \mathtt{b}, \_].\,\mathtt{b}}\ \text{synth/ap}$$

dependent lcf: examples

A more sophisticated example: bidirectional typing rules (not just for typecheckers!—crucial for automation).

$$\frac{\begin{array}{l}[\Omega].\,\Gamma \vdash R\ \mathit{synth} \rightsquigarrow \mathtt{ty} \\ [\Omega, \mathtt{ty}].\,\mathtt{ty}\ \mathit{match}\{0\}\ \mathtt{dfun} \rightsquigarrow \mathtt{a} \\ [\Omega, \mathtt{ty}, \mathtt{a}].\,\mathtt{ty}\ \mathit{match}\{1\}\ \mathtt{dfun} \rightsquigarrow \mathtt{b} \\ [\Omega, \mathtt{ty}, \mathtt{a}, \mathtt{b}].\,S \in \mathtt{a} \rightsquigarrow \_\end{array}}{[\Omega].\,\Gamma \vdash R(S)\ \mathit{synth} \rightsquigarrow [\Omega, \mathtt{ty}, \mathtt{a}, \mathtt{b}, \_].\,\mathtt{b}}\ \text{synth/ap}$$

$$\frac{}{[\Omega].\,\vartheta(M_0, \ldots, M_n)\ \mathit{match}\{i\}\ \vartheta \rightsquigarrow [\Omega].\,M_i}\ \text{match}$$

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

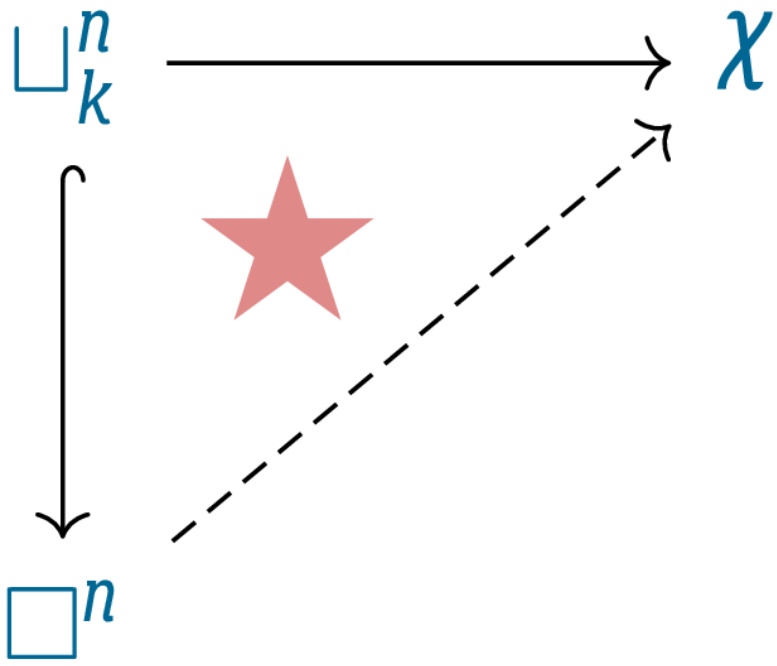
use *dependent lcf* today!

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



CUBICAL THOUGHT IS THE NEVER-SETTING SUN!

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]

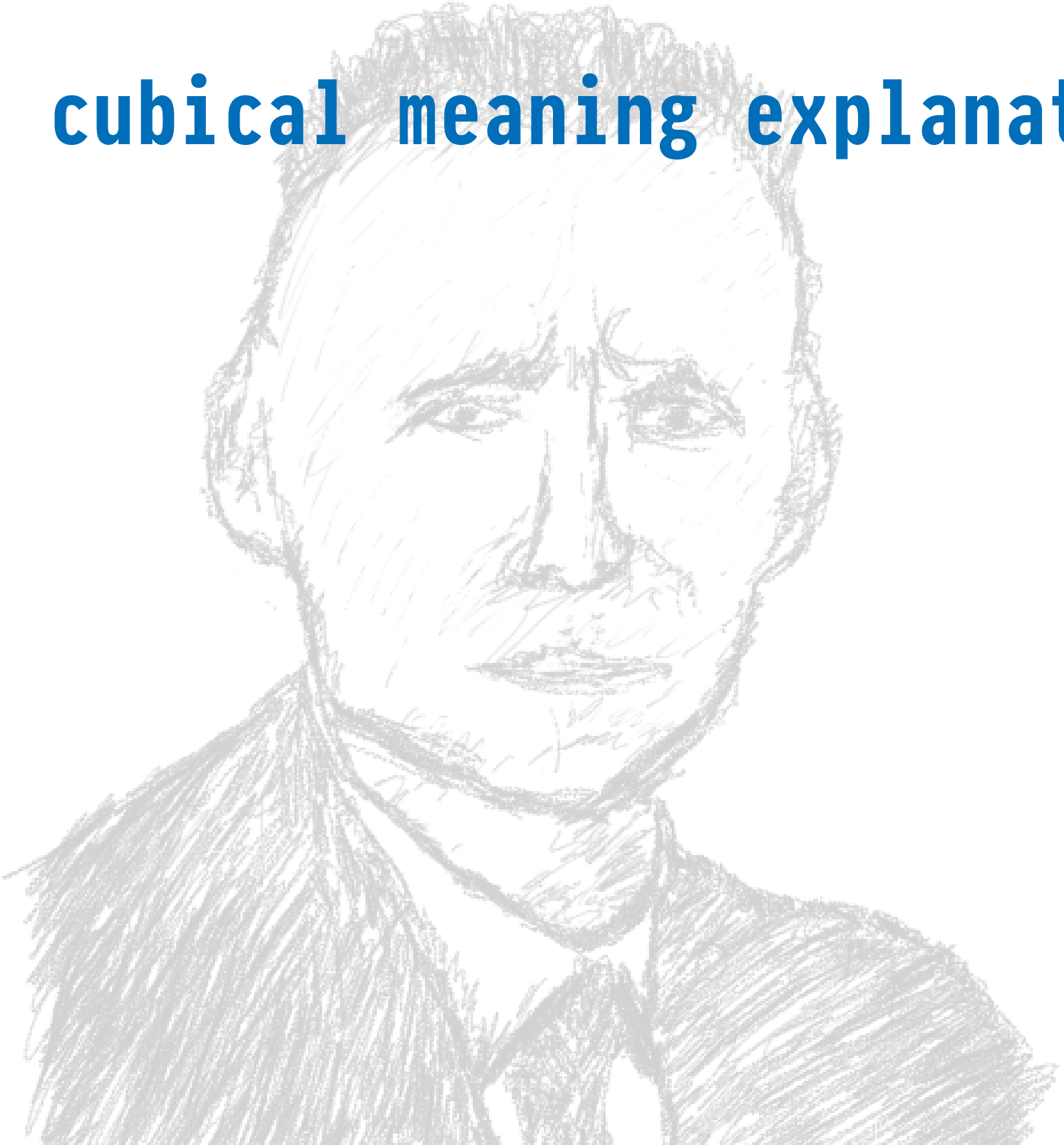
- ⋆ a type theory with both extensional ***equality*** and intensional ***identification*** (paths)
- ⋆ higher inductive types: the circle
- ⋆ strict types: strict booleans (new)
- ⋆ computational canonicity (previous Licata/Harper result established canonicity up to ***judgmental equality*** for 2D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

RedPRL's cubical type theory

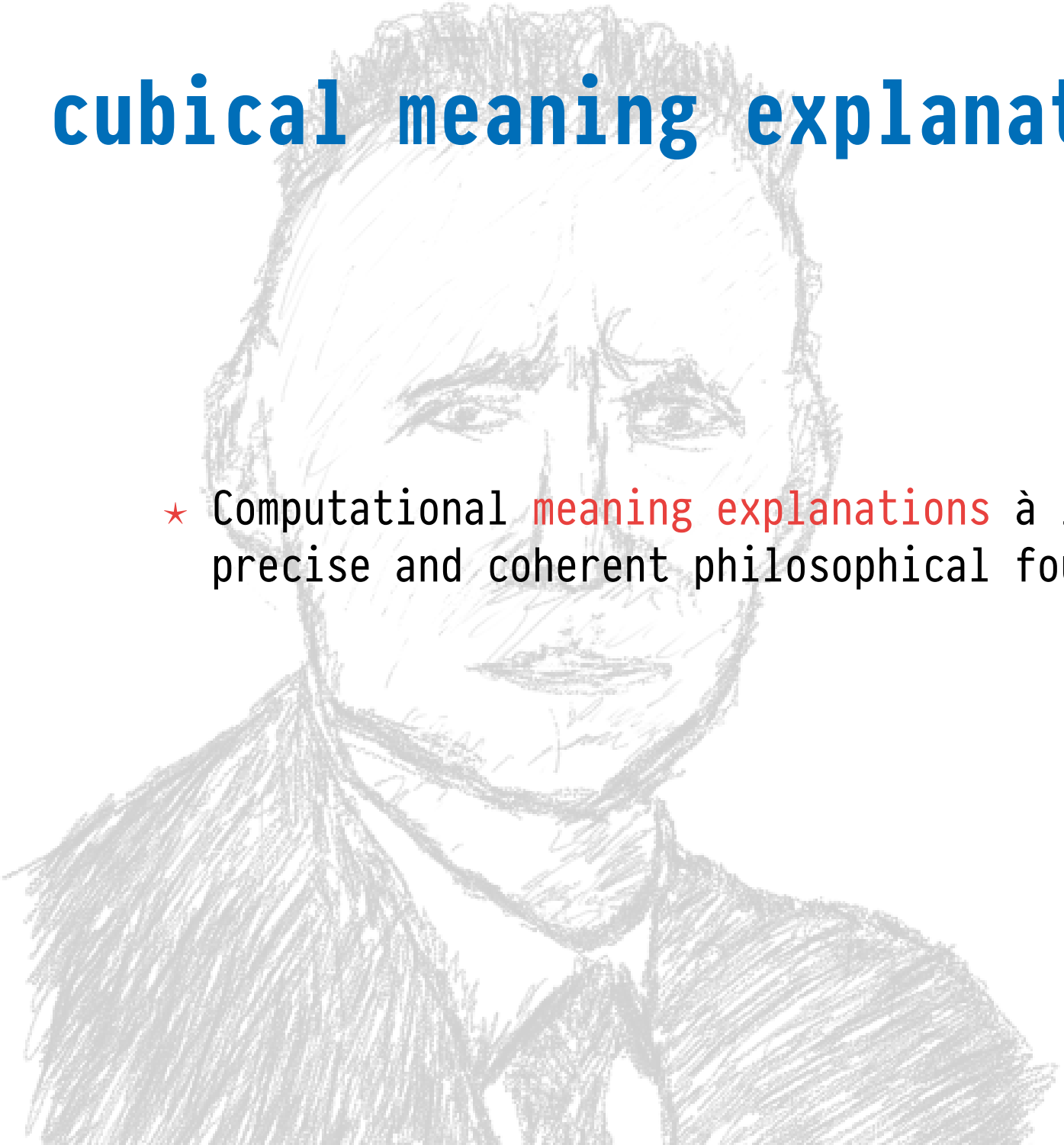
Computational Higher-Dimensional Type Theory
[Angiuli/Harper/Wilson POPL 2017]

- ⋆ 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, not_χ.
- ⋆ a deterministic and type-free operational semantics, amenable to cost analysis.

cubical meaning explanation

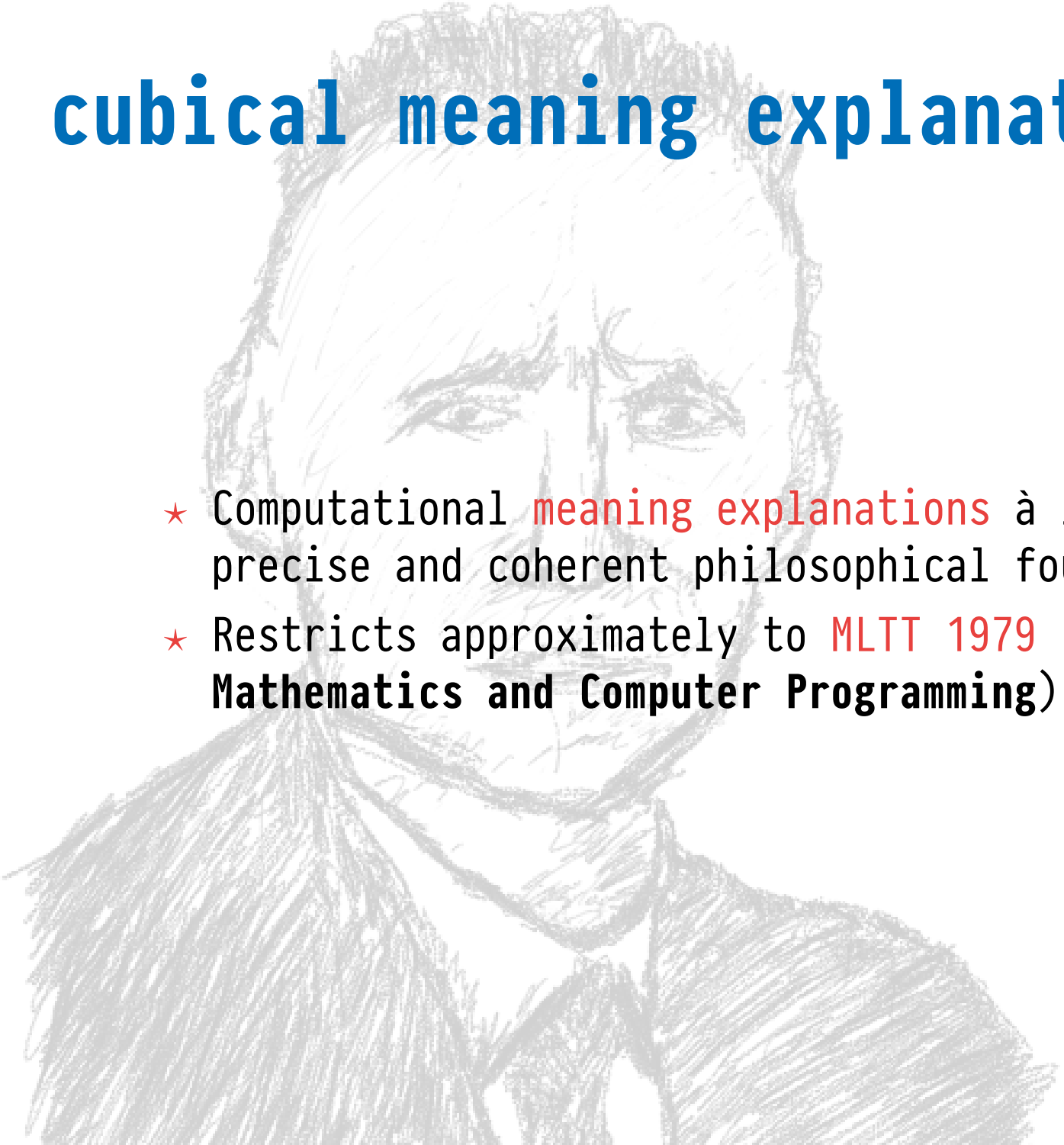


cubical meaning explanation



- ⋆ 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.

future work

- rule calculus
- $\Rightarrow$ MetaPRL-style derived rules
- univalence principle
- computational effects: gensym, exceptions, store, IO
- continuity principles, bar induction


COO