An Attempt at Necromancy

Experience Report on Modernising a Domain Theory Library

Rocqshop – July 25th 2026

Meven Lennon-Bertrand

INRIA – IRIF, Université Paris Cité

of "inclusive predicates", fundamental in domain theory [12, 15]. Another technical advantage of our approach is that we don't need to use contexts as Kripke worlds as in previous arguments [2, 6]. We also establish that two convertible terms in type theory (maybe *partial* [10, 11, 13, 14]) have the same Böhm tree.

In this paper, we work in a constructive metatheory, and when we write that a propoosition P is decidable, we mean that $P \vee \neg P$ is provable.

2 Domain and Finite Elements

We shall use the following Scott domain, least solution of a recursive domain equation (see [18, 19] for a lively description of Scott domains and solutions to domain equations):

$$D \;=\; [D \to D] + \Pi\ D\,[D \to D] + \mathsf{N} + \mathsf{0} + \mathsf{S}\ D + \mathsf{U}_i$$

In this equation, $+$ denotes the coalesced sum [18] and $i = 0, 1, 2, \ldots$

We write $a, b, u, v, \ldots$ for the elements of this domain. We define $u(v)$ for u and v in D as follows: it is the application of u to v if u belongs to $D \to D$ and it is $\perp$ otherwise.

An Adequacy Theorem for Dependent Type Theory (Coquand et al. 2018)

Formalized, Effective Domain Theory in Coq

Robert Dockins

Portland State University, Portland, Oregon, USA
rdockins@pdx.edu

Abstract. I present highlights from a formalized development of domain theory in the theorem prover Coq. This is the first development of domain theory that is *effective*, *formalized* and that supports all the usual constructions on domains. In particular, I develop constructive models of both the unpointed profinite and the pointed profinite domains. Standard constructions (e.g., products, sums, the function space, and powerdomains) are all developed. In addition, I build the machinery necessary to compute solutions to recursive domain equations.

1 Introduction and Related Work

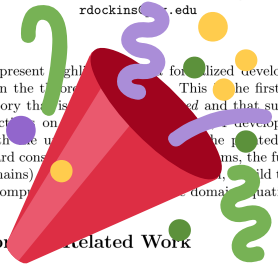
The term "domain theory" refers to a class of mathematical techniques that are used to develop computational models suitable for reasoning about the semantics of general purpose programming languages. Proofs done in domain theory often have an elegant, compositional nature, and the much of the modern thinking about programming languages (especially functional languages) can be traced to domain-theoretic roots. Domain theory has a long lineage, starting from the classic work of Dana Scott on continuous lattices [22]. Domain theory was intensively studied in the 1970's and 80's, producing far more research than I have space here to review; see Abramsky and Jung's account for a good overview and further sources [2].

Unfortunately, using domain theory for language semantics requires special-

Formalized, Effective Domain Theory in Coq

Robert Dockins

Portland State University, Portland, Oregon, USA
rdockins@cs.pdx.edu

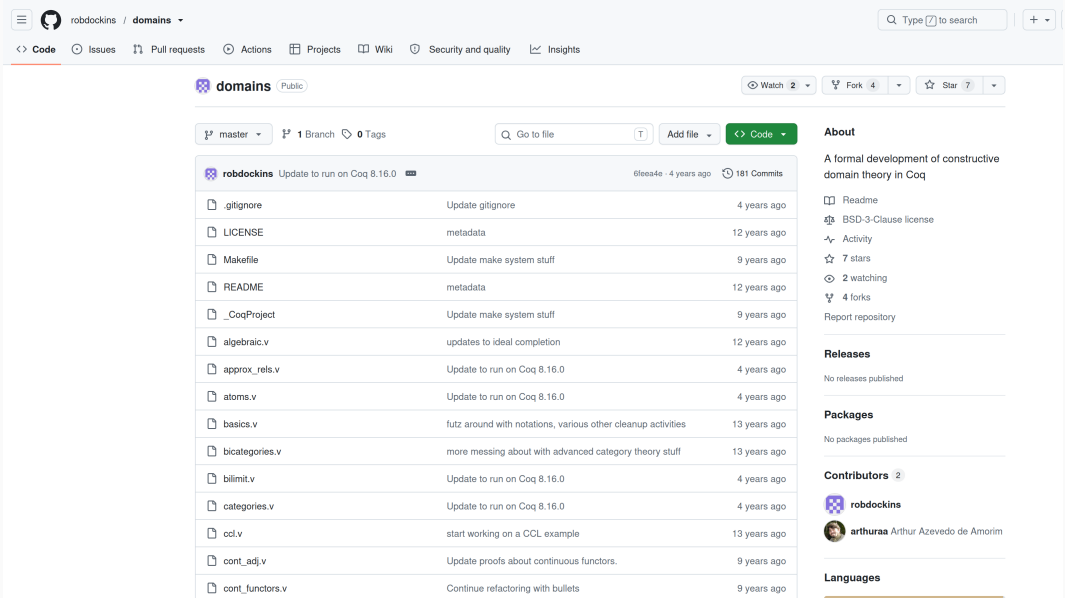


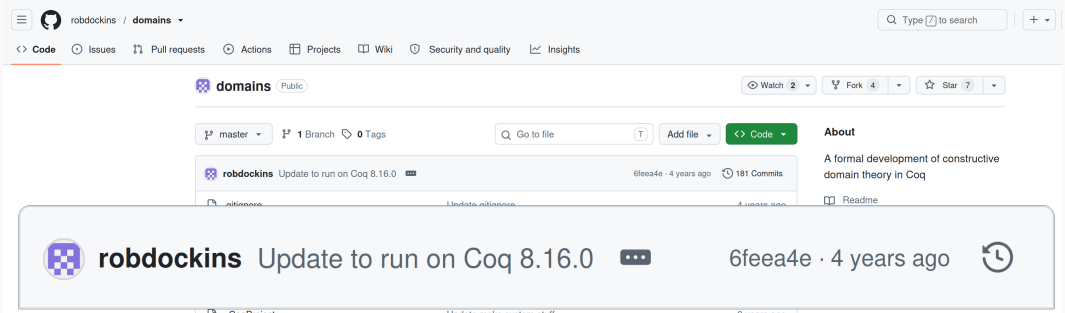
Abstract. I present [illegible] formalized development of domain theory in the [illegible]. This is the first development of domain theory that is [illegible] and that supports all the usual constructions on [illegible]. I develop constructive models of both the [illegible] the pointed profinite domains. Standard constructions [illegible]ms, the function space, and powerdomains) [illegible]ld the machinery necessary to compute [illegible] domain equations.

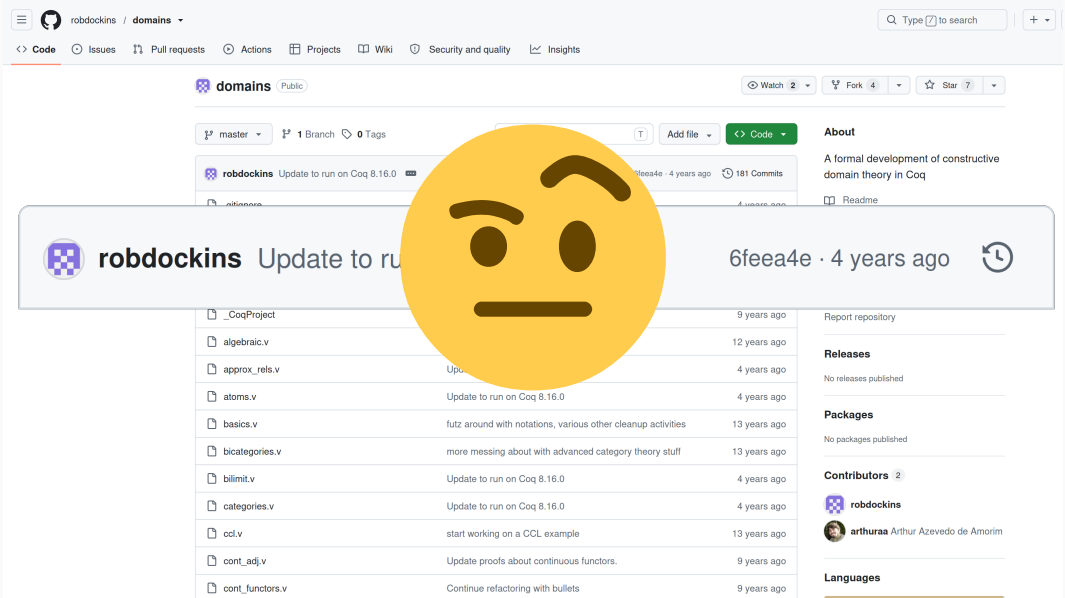
1 Introduction and Related Work

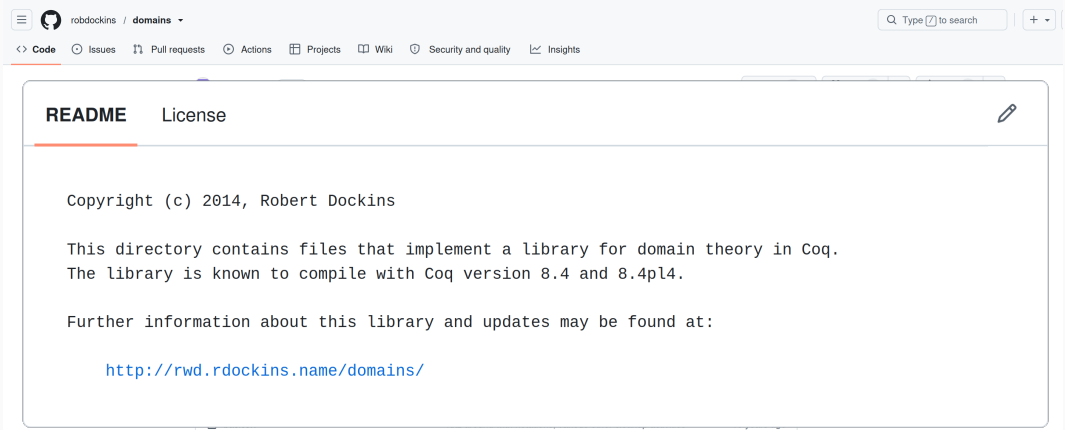
The term "domain theory" refers to a class of mathematical techniques that are used to develop computational models suitable for reasoning about the semantics of general purpose programming languages. Proofs done in domain theory often have an elegant, compositional nature, and the much of the modern thinking about programming languages (especially functional languages) can be traced to domain-theoretic roots. Domain theory has a long lineage, starting from the classic work of Dana Scott on continuous lattices [22]. Domain theory was intensively studied in the 1970's and 80's, producing far more research than I have space here to review; see Abramsky and Jung's account for a good overview and further sources [2].

Unfortunately, using domain theory for language semantics requires special-


robdockins / domains
Type / to search
Code
Issues
Pull requests
Actions
Projects
Wiki
Security and quality
Insights
domains
Public
Watch 2
Fork 4
Star 7
master
1 Branch
0 Tags
Go to file
T
Add file
Code
robdockins Update to run on Coq 8.16.0
6feea4e · 4 years ago
181 Commits
.gitignore Update gitignore 4 years ago
LICENSE metadata 12 years ago
Makefile Update make system stuff 9 years ago
README metadata 12 years ago
_CoqProject Update make system stuff 9 years ago
algebraic.v updates to ideal completion 12 years ago
approx_rels.v Update to run on Coq 8.16.0 4 years ago
atoms.v Update to run on Coq 8.16.0 4 years ago
basics.v futz around with notations, various other cleanup activities 13 years ago
bicategories.v more messing about with advanced category theory stuff 13 years ago
bilimit.v Update to run on Coq 8.16.0 4 years ago
categories.v Update to run on Coq 8.16.0 4 years ago
ccl.v start working on a CCL example 13 years ago
cont_adj.v Update proofs about continuous functors. 9 years ago
cont_functors.v Continue refactoring with bullets 9 years ago
About
A formal development of constructive domain theory in Coq
Readme
BSD-3-Clause license
Activity
7 stars
2 watching
4 forks
Report repository
Releases
No releases published
Packages
No packages published
Contributors 2
robdockins
arthuraa Arthur Azevedo de Amorim
Languages


robdockins / domains
Type / to search
Code
Issues
Pull requests
Actions
Projects
Wiki
Security and quality
Insights
domains
Public
Watch 2
Fork 4
Star 7
master
1 Branch
0 Tags
Go to file
Add file
Code
robdockins Update to run on Coq 8.16.0
6feea4e · 4 years ago
181 Commits
About
A formal development of constructive domain theory in Coq
Readme
robdockins Update to run on Coq 8.16.0
6feea4e · 4 years ago
_CoqProject Update make system stuff 9 years ago
algebraic.v updates to ideal completion 12 years ago
approx_rels.v Update to run on Coq 8.16.0 4 years ago
atoms.v Update to run on Coq 8.16.0 4 years ago
basics.v futz around with notations, various other cleanup activities 13 years ago
bicategories.v more messing about with advanced category theory stuff 13 years ago
bilimit.v Update to run on Coq 8.16.0 4 years ago
categories.v Update to run on Coq 8.16.0 4 years ago
ccl.v start working on a CCL example 13 years ago
cont_adj.v Update proofs about continuous functors. 9 years ago
cont_functors.v Continue refactoring with bullets 9 years ago
Report repository
Releases
No releases published
Packages
No packages published
Contributors 2
robdockins
arthuraa Arthur Azevedo de Amorim
Languages


robdockins / domains
Type / to search
Code
Issues
Pull requests
Actions
Projects
Wiki
Security and quality
Insights
domains
Public
Watch 2
Fork 4
Star 7
master
1 Branch
0 Tags
Add file
Code
robdockins Update to run on Coq 8.16.0
6feea4e · 4 years ago
181 Commits
robdockins Update to ru
6feea4e · 4 years ago
_CoqProject 9 years ago
algebraic.v 12 years ago
approx_rels.v 4 years ago
atoms.v Update to run on Coq 8.16.0 4 years ago
basics.v futz around with notations, various other cleanup activities 13 years ago
bicategories.v more messing about with advanced category theory stuff 13 years ago
bilimit.v Update to run on Coq 8.16.0 4 years ago
categories.v Update to run on Coq 8.16.0 4 years ago
ccl.v start working on a CCL example 13 years ago
cont_adj.v Update proofs about continuous functors. 9 years ago
cont_functors.v Continue refactoring with bullets 9 years ago
About
A formal development of constructive domain theory in Coq
Readme
Report repository
Releases
No releases published
Packages
No packages published
Contributors 2
robdockins
arthuraa Arthur Azevedo de Amorim
Languages



bicategories.v	more messing about with advanced category theory stuff	13 years ago
bilimit.v	Update to run on Coq 8.16.0	4 years ago
categories.v	Update to run on Coq 8.16.0	4 years ago
ccl.v	start working on a CCL example	13 years ago
cont_adj.v	Update proofs about continuous functors.	9 years ago
cont_functors.v	Continue refactoring with bullets	9 years ago

No packages published

Contributors 2

robdockins

arthuraa Arthur Azevedo de Amorim

Languages

robdockins / domains

Type / to search

Code Issues Pull requests Actions Projects Wiki Security and quality Insights

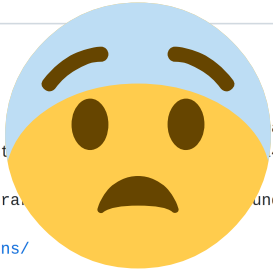
README License

Copyright (c) 2014, Robert Dockins

This directory contains files that ain theory in Coq.
The library is known to compile wit 4.

Further information about this libra und at:

http://rwd.rdockins.name/domains/



bicategories.v	more messing about with advanced category theory stuff	13 years ago
bilimit.v	Update to run on Coq 8.16.0	4 years ago
categories.v	Update to run on Coq 8.16.0	4 years ago
ccl.v	start working on a CCL example	13 years ago
cont_adj.v	Update proofs about continuous functors.	9 years ago
cont_functors.v	Continue refactoring with bullets	9 years ago

No packages published

Contributors 2

robdockins

arthuraa Arthur Azevedo de Amorim

Languages

The plan

1. Some reflexions on porting
2. Some technical reflexions

The plan

1. **Some reflexions on porting**

2. **Some technical reflexions**

More food for thought than very conclusive
⟶ Let's chat!

Some reflexions on porting

Coq 8.16 → Rocq 9.1

It was a breeze!

Coq 8.16 → Rocq 9.1

It was a breeze! But...

```
destruct (finsubset_dec' A (OrdDec A (eff_ord_dec A Heff)) P') with M; auto.
- intro x. unfold P'.
  destruct (inh_dec A hf x).
  + destruct (finset_find_dec' A (fun p:A ⇒ p ∈ M)) with x; simpl.
    * intros. rewrite <- H0; auto.
    * intros. apply finset_in_dec.
      constructor. apply eff_ord_dec. auto.
    * left. intros Hx ?.
      destruct s. destruct a.
      apply H0 in H1. elim H2; auto.
    * destruct (normal_has_mubs Q HQ x) as [MUBS [?[??]]]; auto.
      ** red; intros. apply HM. apply m. auto.
      ** destruct (finset_find_dec' A (fun p ⇒ p ∈ M)) with MUBS; simpl.
          [ ... ]
          *** left. intros _. intros.
              apply m0.
              destruct (H2 x0) as [x0' [?[??]]].
              **** destruct H4; auto.
```

Rocq 9.1 → Rocq 9.1

What does it mean to “port a library to Rocq 9.1”?

1. it compiles?
2. it is idiomatic?

Rocq 9.1 → Rocq 9.1

What does it mean to “port a library to Rocq 9.1”?

1. it compiles?
2. it is idiomatic?

- tactics & automation
- inference & structures (typeclasses/canonical structures/etc.)
- foundations?
- ...

Rocq 9.1 → Rocq 9.1

What does it mean to “port a library to Rocq 9.1”?

1. it compiles?
2. it is idiomatic?

- tactics & automation
- inference & structures (typeclasses/canonical structures/etc.)
- foundations?
- ...

Libraries will decay

Rocq 9.1 → Rocq 9.1

What does it mean to “port a library to Rocq 9.1”?

1. it compiles?
2. it is idiomatic?

- tactics & automation
- inference & structures (typeclasses/canonical structures/etc.)
- foundations?
- ...

Libraries will decay

Is long-term backwards compatibility that important?

Theory building

Programming as Theory Building (Naur 1985)

Theory building

A person who has or possesses a theory in this sense knows how to do certain things and in addition can support the actual doing with explanations, justifications, and answers to queries, about the activity of concern.
[...]
The programmer having the theory of the program can explain why each part of the program is what it is, in other words is able to support the actual program text with a justification of some sort.

Programming as Theory Building (Naur 1985)

A person who has or possesses a theory in this sense knows how to do certain things and in addition can support the actual doing with explanations, justifications, and answers to queries, about the activity of concern.
[...]
The programmer having the theory of the program can explain why each part of the program is what it is, in other words is able to support the actual program text with a justification of some sort.
[...]
The death of a program happens when the programmer team possessing its theory is dissolved. A dead program may continue to be used for execution in a computer and to produce useful results. [...] Revival of a program is the rebuilding of its theory by a new programmer.

Programming as Theory Building (Naur 1985)

Theory building

A person who has or possesses a theory in this sense knows how to do certain things and in addition can support the actual doing with explanations, justifications, and answers to queries, about the activity of concern.
[...]
The programmer having the theory of the program can explain why each part of the program is what it is, in other words is able to support the actual program text with a justification of some sort.
[...]
The death of a program happens when the programmer team possessing its theory is dissolved. A dead program may continue to be used for execution in a computer and to produce useful results. [...] **Revival of a program is the rebuilding of its theory** *by a new programmer.*

Programming as Theory Building (Naur 1985)

Please don't tell me about AI

An attempt at necromancy

The new programmer is likely to feel torn between loyalty to the existing program text, with whatever obscurities and weaknesses it may contain, and the new theory that [they have] to build up, and which [...] will differ from the original theory behind the program text.

An attempt at necromancy

> *The new programmer is likely to feel torn between loyalty to the existing program text, with whatever obscurities and weaknesses it may contain, and the new theory that [they have] to build up, and which [...] will differ from the original theory behind the program text.*

Concretely: is this design decision

- good (although maybe I can't appreciate why)?
- bad (and I can/should change it)?
- historically good but not valid any more?

What to keep?

In preference to program revival, the Theory Building View suggests, the existing program text should be discarded and the new-formed programmer team should be given the opportunity to solve the given problem afresh.

What to keep?

> *In preference to program revival, the Theory Building View suggests, the existing program text should be discarded and the new-formed programmer team should be given the opportunity to solve the given problem afresh.*

A bit extreme?
My approach: original mechanisation = very detailed mathematical text to mechanise

- mostly keep mathematical definitions
- change the infra
- re-do the proofs

What to keep?

> *In preference to program revival, the Theory Building View suggests, the existing program text should be discarded and the new-formed programmer team should be given the opportunity to solve the given problem afresh.*

A bit extreme?
My approach: original mechanisation = very detailed mathematical text to mechanise

- mostly keep mathematical definitions
- change the infra
- re-do the proofs

It's nice to run the original proof script

The documentation problem

The knowledge possessed by the programmer by virtue of [them] having the theory necessarily [...] transcends that which is recorded in the documented products.

The documentation problem

> *The knowledge possessed by the programmer by virtue of [them] having the theory necessarily [...] transcends that which is recorded in the documented products.*

I don't (fully) agree with Naur: mechanisations = very specific "programs"

- we write papers
- the point of a mechanisation is communication

The documentation problem

> *The knowledge possessed by the programmer by virtue of [them] having the theory necessarily [...] transcends that which is recorded in the documented products.*

I don't (fully) agree with Naur: mechanisations = very specific "programs"

- we write papers
- the point of a mechanisation is communication

Still: no record of the theory ⟶ bad long-term decay

The documentation problem

> *The knowledge possessed by the programmer by virtue of [them] having the theory necessarily [...] transcends that which is recorded in the documented products.*

I don't (fully) agree with Naur: mechanisations = very specific "programs"

- we write papers
- the point of a mechanisation is communication

Still: no record of the theory ⟶ bad long-term decay

Think about this when mechanising & writing/reviewing papers!

Some technical reflexions

Setoids, setoids everywhere

```
Record theory (set : preord -> Set) := {
member : forall (A:preord), A -> set A -> Prop.
member_eq : forall (A:preord) (a b:A) (X:set A),
    a ≈ b -> member A a X -> member A b X;
… }.

Definition fset (A:preord) := list A.
Definition fmember (A:preord) (a:A) (P:fset A) := exists a', In a' P ∧ a ≈ a'.
```

Setoids, setoids everywhere

```
Record theory (set : preord -> Set) := {
member : forall (A:preord), A -> set A -> Prop.
member_eq : forall (A:preord) (a b:A) (X:set A),
    a ≈ b -> member A a X -> member A b X;
… }.

Definition fset (A:preord) := list A.
Definition fmember (A:preord) (a:A) (P:fset A) := exists a', In a' P ∧ a ≈ a'.
```

Solution = `Proper` + `setoid rewrite`?

- still quite bureaucratic
- type-classes vs canonical structures
- `setoid rewrite` is not as good as `rewrite`

Setoids, setoids everywhere

```
Record theory (set : preord -> Set) := {
member : forall (A:preord), A -> set A -> Prop.
member_eq : forall (A:preord) (a b:A) (X:set A),
    a ≈ b -> member A a X -> member A b X;
… }.

Definition fset (A:preord) := list A.
Definition fmember (A:preord) (a:A) (P:fset A) := exists a', In a' P ∧ a ≈ a'.
```

Solution = `Proper` + `setoid rewrite`?

- still quite bureaucratic
- type-classes vs canonical structures
- `setoid rewrite` is not as good as `rewrite`

Synthetic setoids

Synthetic setoids

Axiomatised extensionality features:

- funext
- propext + proof irrelevance
- quotients

Justified by an easy setoid model!

Synthetic setoids

Axiomatised extensionality features:

- funext
- propext + proof irrelevance
- quotients

Justified by an easy setoid model!

The model handles the bureaucracy…
except when it's not bureaucracy, *e.g.* quotients

Synthetic setoids

Axiomatised extensionality features:

- funext
- propext + proof irrelevance
- quotients

Justified by an easy setoid model!

The model handles the bureaucracy…
except when it's not bureaucracy, *e.g.* quotients

Small tactic computes identity types:

```
f, g : nat -> {n : nat & P n}
═══════════════════════════════
f = g
```

->

```
f, g : nat -> {n : nat & P n}
x : nat
═══════════════════════════════
proj1_sig (f x) = proj1_sig (g x)
```

Choosing is always a problem

Vanilla setoid model validates little choice...

Choosing is always a problem

Vanilla setoid model validates little choice...

Issue 1: Less flexibility than with (analytic) setoids

Countable union of countable sets = fine with setoids, countable choice synthetically

Valid in the model, added as **axiom**

Choosing is always a problem

Vanilla setoid model validates little choice...

Issue 1: Less flexibility than with (analytic) setoids

Countable union of countable sets = fine with setoids, countable choice synthetically

Valid in the model, added as **axiom**

Issue 2: Unique choice

Two truncations, two existentials

Invalid in the model!

Choosing is always a problem

Vanilla setoid model validates little choice...

Issue 1: Less flexibility than with (analytic) setoids

Countable union of countable sets = fine with setoids, countable choice synthetically

Valid in the model, added as **axiom**

Issue 2: Unique choice

Two truncations, two existentials

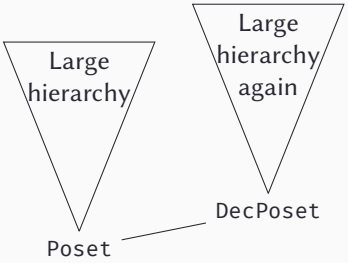
Invalid in the model!

Is there a better model? Is it worth losing the intuitive setoid reading?

Hierarchy builder is very very nice!

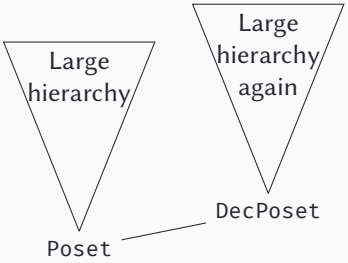
Hierarchy builder is very very nice! When it works...

Hierarchy builder is very very nice! When it works...



Is there a lighter way?

Hierarchy builder is very very nice! When it works...



Is there a lighter way?

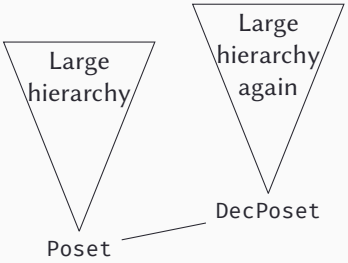
"Complicated" inference patterns:

```
Record IsDecPoset (T : Type) of poset T :=
  { dec (a b : T), IsDecidable (a ⩽ b) }.

Record IsGroupoid (T : Type) of quiver T :=
  { inv (a b : T) (f : hom a b), IsInvertible f }.

Record SetTheory :=
  { set : Type -> Poset ; … }.
```

Hierarchy builder is very very nice! When it works...



Is there a lighter way?

"Complicated" inference patterns:

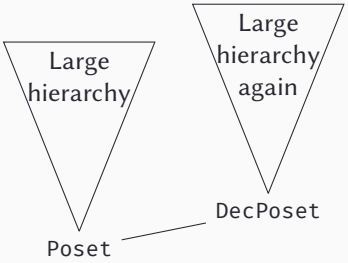
```
Record IsDecPoset (T : Type) of poset T :=
  { dec (a b : T), IsDecidable (a ⩽ b) }.

Record IsGroupoid (T : Type) of quiver T :=
  { inv (a b : T) (f : hom a b), IsInvertible f }.

Record SetTheory :=
  { set : Type -> Poset ; … }.
```

Type-classes ? Some sort of combination? A new mechanism?

Hierarchy builder is very very nice! When it works...



Is there a lighter way?

“Complicated” inference patterns:

```
Record IsDecPoset (T : Type) of poset T :=
  { dec (a b : T), IsDecidable (a ⩽ b) }.

Record IsGroupoid (T : Type) of quiver T :=
  { inv (a b : T) (f : hom a b), IsInvertible f }.

Record SetTheory :=
  { set : Type -> Poset ; … }.
```

Type-classes ? Some sort of combination? A new mechanism?

Also, in practice: lots of layers that can go wrong (and do...)

Wrapping up

Go read *Programming as Theory Building*

The future is coming, but there's still some way to go

Go read *Programming as Theory Building*

The future is coming, but there's still some way to go

Thank you!