A formalization of forcing and the unprovability of the continuum hypothesis

Jesse Michael Han

ITP 2019

University of Pittsburgh

joint w/ Floris van Doorn

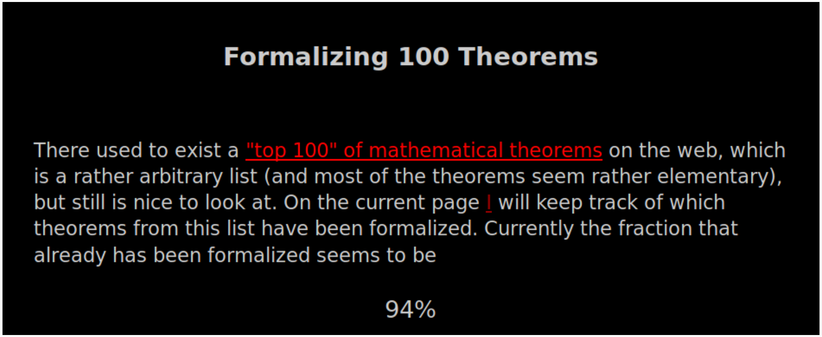
Outline

Continuum hypothesis

- Posed by Cantor in 19th century: does there exist an infinite cardinality strictly larger than the countable natural numbers $\mathbb{N}$ but strictly smaller than the uncountable real numbers $\mathbb{R}$?
- was Hilbert's 1st question
- Proved independent (neither provable nor disprovable) from ZFC by Paul Cohen ('60s) and Kurt Godel ('30s). Cohen's invention of forcing earned him a Fields medal, the only one ever awarded for work in mathematical logic.

Continuum hypothesis

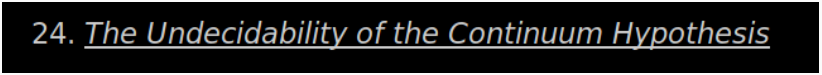
- Independence of CH has never been formalized!



Formalizing 100 Theorems

There used to exist a "top 100" of mathematical theorems on the web, which is a rather arbitrary list (and most of the theorems seem rather elementary), but still is nice to look at. On the current page I will keep track of which theorems from this list have been formalized. Currently the fraction that already has been formalized seems to be

94%



24. *The Undecidability of the Continuum Hypothesis*

The Flypitch project

FLYPITCH

formally proving the independence of the continuum hypothesis

Website: flypitch.github.io

- Aims to formalize the independence of CH
- Along the way, build reusable libraries for mathematical logic and set theory
- Written in Lean 3.

What is required for the formalization?

To formalize just the statement, "the continuum hypothesis is neither provable nor disprovable from ZFC", we need:

- First-order logic (terms, formulas, quantifiers, sentences. . .)
- Provability, i.e. a proof system
- The axioms of ZFC and also CH as first-order formulas

To formalize the proof, we need:

- Semantics (ordinary soundness theorem)
- Completeness theorem
- Boolean-valued semantics and soundness for first-order logic
- Forcing

First-order logic

```
structure Language : Type (u+1) :=
    (functions : ℕ → Type u)
    (relations : ℕ → Type u)

/- The language of abelian groups -/
inductive abel_functions : ℕ → Type
| zero : abel_functions 0
| plus : abel_functions 2

def L_abel : Language := ⟨abel_functions, (λ _, empty)⟩
```

First-order logic

```
inductive preterm : ℕ → Type u
| var : ∀ (k : ℕ), preterm 0 -- notation &
| func : ∀ {l : ℕ} (f : L.functions l), preterm l
| app : ∀ {l : ℕ} (t : preterm (l + 1)) (s : preterm 0),
    preterm l

def term := preterm L 0
```

- `preterm L n` is a partially applied term. If applied to `n` terms, it becomes a `term`.
- Every element of `preterm L 0` is a well-formed term.
- We use this encoding to avoid mutual or nested inductive types, since those are not too convenient to work with in Lean.

First-order logic

Similarly for formulas:

```
inductive preformula : ℕ → Type u
| falsum {} : preformula 0 -- notation ⊥
| equal (t₁ t₂ : term L) : preformula 0 -- notation ≃
| rel {l : ℕ} (R : L.relations l) : preformula l
| apprel {l : ℕ} (f : preformula (l + 1)) (t : term L) :
    preformula l
| imp (f₁ f₂ : preformula 0) : preformula 0 -- notation ⟹
| all (f : preformula 0) : preformula 0 -- notation ∀'

def formula := preformula L 0
```

A model of ZFC in Lean

The following construction is due to Aczel, and was implemented in Coq by Werner and then in Lean by Carneiro:

```
inductive pSet : Type (u+1)
| mk (α : Type u) (A : α → pSet) : pSet
```

- Note that `mk empty empty.elim` always exists, and corresponds to the empty set at the bottom of the von Neumann hierarchy.
- (Extensional) equality can be defined by structural recursion (the elimination principle for the inductive type `pSet` *is* $\in$-recursion): Two pre-sets are extensionally equivalent if every element of the first family is extensionally equivalent to some element of the second family and vice-versa.

A model of ZFC in Lean

```
def equiv : ∀ (x y : pSet), Prop
| ⟨α,A⟩ ⟨β,B⟩ := (∀a : α, ∃b : β, equiv (A a) (B b)) ∧ (∀b : β
    , ∃a : α, equiv (A a) (B b))
```

- Notice that an equivalent model of ZFC is produced by defining:

```
inductive bSet : Type (u+1)
| mk (α : Type u) (A : α → bSet) (A′ : α → bool) : bSet
```

- At first, this seems needlessly complicated (we can recover the original `pSet` by recursively ignoring anything which is assigned `ff : bool`).

A model of ZFC in Lean

- However, if we remember that `bool` is a complete boolean algebra $(\text{bool}, \top, \bot, \neg, \sqcup, \sqcap, \bigsqcup, \bigsqcap, \Rightarrow)$, we see that by replacing universal/existential quantification in `Prop` with infima/suprema in `bool`, we can "internalize" the truth-values of `equiv₂` to `bool`:

```
def equiv₁ : ∀ (x y : bSet), Prop
| ⟨α,A,A′⟩ ⟨β,B,B′⟩ := (∀ a : α, A′ a = tt → ∃ b : β, B′ b = tt ∧
     equiv₁ (A a) (B b)) ∧ (∀b : β, B′ b = tt → ∃ a : α, A′ a =
     tt ∧ equiv₁ (A a) (B b))

/- Assuming Prop ≃ bool, equiv₂ is equivalent to equiv₁-/
def equiv₂ : ∀ (x y : bSet), bool
| ⟨α,A,A′⟩ ⟨β,B,B′⟩ := ⨅ a : α, A′ a ⟹ ⨆ b : β, B′ b ⊓ equiv₂
    (A a) (B b) ⊓  ⨅ b : β, B′ b ⟹ ⨆ a : α, A′ a ⊓ equiv₂
    (A a) (B b)
```

The forcing extension

The definition of `equiv₂` makes sense for any complete boolean algebra.

Accordingly, we define

```
inductive bSet (𝔹 : Type u) [complete_boolean_algebra 𝔹] : Type
    (u+1)
| mk (α : Type u) (A : α → bSet) (B : α → 𝔹) : bSet
```

Note:

- When 𝔹 is the singleton algebra `unit`, `bSet unit` is isomorphic to `pSet`.
- `bSet 𝔹` is exactly the "name construction" from forcing; `bSet 𝔹` comprises the "𝔹-names".

Theorem. For every 𝔹, `bSet 𝔹` is a **boolean-valued model** of ZFC.

Boolean-valued semantics

- In general, can define $\mathbb{B}$-valued semantics for arbitrary first-order theories and prove a soundness theorem (if ϕ has truth-value $b \in \mathbb{B}$ in some $\mathbb{B}$-valued model $\mathbf{M}$ and $\phi \vdash \psi$, then ψ has truth-value greater than or equal to b).
- Equality is interpreted as binary $\mathbb{B}$-valued function $(=^{\mathbb{B}})$.
- An n-ary relation symbol is interpreted as an n-ary $\mathbb{B}$-valued function satisfying a $\mathbb{B}$-valued congruence lemma with respect to equality (e.g. $x =^{\mathbb{B}} y \sqcap R(x) \leqslant R(y)$).

Boolean-valued semantics

- Helpful special case: $\mathbb{B} =$ (measure algebra of a probability space modulo measure-zero events).
- It is profitable to keep in mind the following analogy, developed by Scott. Let $\mathbf{M}$ be a $\mathbb{B}$-valued structure.
- A unary $\mathbb{B}$-valued predicate ϕ on $\mathbf{M}$ assigns an event to every element m of $\mathbf{M}$, whose measure we can think of as being the probability that $\phi(m)$ is true.
- Specializing to the language of set theory, in a $\mathbb{B}$-valued model of set theory, set membership $(X \overset{?}{\in} Y)$ is no longer bool-valued, but $\mathbb{B}$-valued. Subsets of Z are determined by indicator functions $\chi : Z \to \mathbb{B}$.

Boolean-valued semantics

- To every $\mathbb{B}$-valued set $(m : \mathbf{M})$, we can attach an "indicator function" $\lambda x, x \in m$ which assigns to every x a probability that it is actually a member of m.
- Thus, by virtue of extensionality, we may think of the elements of a $\mathbb{B}$-valued model of ZFC as being "set-valued random variables", or "random sets".
- (In this analogy, given a universe of random sets, the purpose of the generic filter or ultrafilter in forcing is then to simultaneously evaluate the outcomes of the random variables, collapsing them into an ordinary universe of sets.)

Unprovability of CH

- How do we represent CH as a formula in ZFC?
  - Let $x \prec y$ mean "there is no surjection from x onto y"
  - Let $x \preceq y$ mean "there is an injection from x into y"
  - Then

$$\mathrm{CH} := \neg \left(\exists x \exists y, (\omega \prec x) \wedge (x \prec y) \wedge (y \preceq \mathcal{P}(\omega)) \right).$$

- To show ZFC $\nvdash$ CH, suffices (by applying the $\mathbb{B}$-valued soundness theorem) to exhibit a $\mathbb{B}$-valued model of ZFC such that CH has truth value $\neq \top$.
- So, it remains to find some complete boolean algebra $\mathbb{B}$ such that CH is not true in `bSet` $\mathbb{B}$.

Forcing

- Traditionally, forcing is thought of as a technique for extending a *ground model* of ZFC to a *forcing extension* (another model of ZFC), where certain properties can be "forced" to be true or false, and the complete boolean algebra $\mathbb{B}$ usually lives in the ground model.
- Our situation is different. `pSet` can be thought of as a "standard model" of ZFC in Lean, but it is built from a type universe `Type u`, and generally the parameter $\mathbb{B}$ (in `bSet 𝔹`) will be a type in `Type u`.
- The "ground model" (`pSet`) still embeds into the forcing extension: for any $\mathbb{B}$, there is a canonical map ("check-names") from `pSet` to `bSet 𝔹`:

```
-- notation (x ↦ x̌)
def check : (pSet : Type (u+1)) → bSet 𝔹
| ⟨α,A⟩ := ⟨α, λ a, check (A a), λ a, ⊤⟩
```

Forcing

- To perform forcing, we need to understand how the metatheory (Lean) interacts with the internal logic of the forcing extension (`bSet` $\mathbb{B}$).
- These interactions are mediated by `check : pSet → bSet` $\mathbb{B}$.
- The ordinals attached to `Type u` are equivalence classes of well-ordered types in `Type u`, thus `ordinal.(u) : Type (u+1)`.
- The internal logic of `pSet` is closely tied to that of Lean: the class of set-theoretic ordinals in `pSet` is isomorphic to `ordinal.(u)`.
- In turn, `check` sends `pSet` ordinals into the `bSet` ordinals.
- Note: ω in `bSet` is equal to $\check{\omega}$

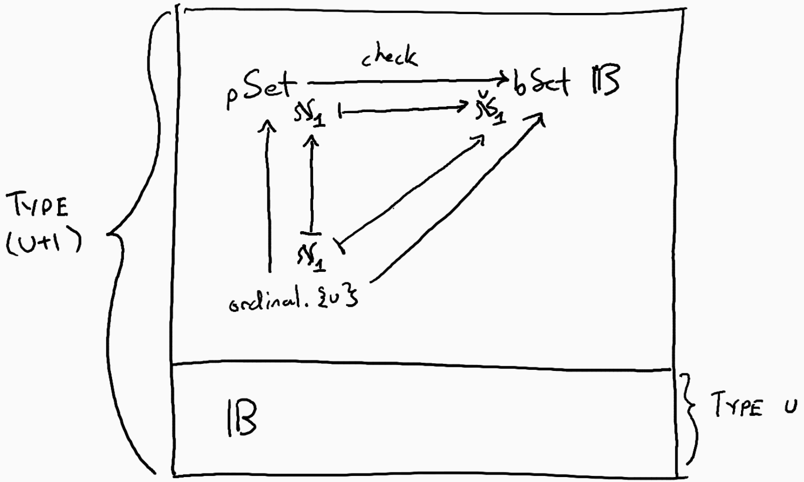
Forcing

The isomorphism between Lean ordinals and pSet ordinals is given by indexing the construction of von Neumann ordinals in one direction, and sending a set to its ordinal rank in the other:

```
-- pseudocode
def pSet.ordinal.mk : ordinal → pSet
| 0 := ∅
| succ ξ := pSet.succ (ordinal.mk ξ) -- (mk ξ ∪ {mk ξ})
| is_limit ξ := ⋃ η < ξ, (ordinal.mk η)


def rank : pSet → ordinal
| ⟨α, A⟩ := ordinal.sup (λ a : α, (rank (A a)).succ)
```

Forcing



Forcing

- We want to pick some $\mathbb{B}$ such that

$$\neg\mathrm{CH} := \exists x \exists y, (\omega \prec x) \wedge (x \prec y) \wedge (y \preceq \mathcal{P}(\omega))$$

is true.

- The idea is that we pick x to be $\check{\aleph}_1$ and y to be $\check{\aleph}_2$.

We need to ensure two things:

1. that there are no surjections from ω onto $\check{\aleph}_1$ or $\check{\aleph}_1$ onto $\check{\aleph}_2$, and
2. that there is an injection from $\check{\aleph}_2$ into $\mathcal{P}(\omega)$.

Converting internal ∀∃ statements to external ∀∃ statements

Let `(x,y : pSet)` such that $x \prec y$ and suppose that `bSet` $\mathbb{B}$ thinks there is a surjection f from $\check{x}$ onto $\check{y}$. That is, in $\mathbb{B}$, we have that

$\top \leqslant$ `(is_func f)` $\sqcap$ `(`$\bigsqcap$ `z` $\in$ `y,` $\bigsqcup$ `w` $\in$ `x, f` $\check{\texttt{w}}$ $=^{\mathbb{B}}$ $\check{\texttt{z}}$`)`

Then the key observation is that this ∀∃-statement in `bSet` $\mathbb{B}$ can be turned in to a ∀∃-statement in the metatheory:

$\forall$ `(j : y.type),` $\exists$ `(i : x.type),` $\bot <$ `(is_func f)` $\sqcap$ `f (x.func i)`$\check{}$
    $=^{\mathbb{B}}$ `(y.func j)`$\check{}$

A mild combinatorial condition on $\mathbb{B}$ ensuring that all antichains are countable (CCC) rules out the above statement from being true when `y.type` is uncountable (because then, any function induced by the ∀∃ statement above has an uncountable fiber indexing an antichain).

Cohen forcing

- A subset of a topological space X is regular open if it is the interior of its closure.
- The regular opens $RO(X)$ form a complete boolean algebra.
- The boolean algebra we use to force ¬CH is:

$$\mathbb{B}\texttt{\_cohen} := \mathrm{RO}(2^{\aleph_2 \times \omega})$$

- X has a basis of clopens, generated by a subbasis of "principal opens" parametrized by $\aleph_2 \times \omega$ of the form

```
principal_open (ν, n) := { g : ℵ₂ × ω → 2 | g (ν, n) = 0}
```

Cohen forcing

- Recall that for every $\mathbb{B}$, the powerset of ω in `bSet 𝔹` is determined by indicator functions $\chi : \omega \to \mathbb{B}$. So, by virtue of our choice of `𝔹 := 𝔹_cohen`, for every $\nu < \aleph_2$, we can attach the following indicator function:

```
def cohen_real.mk : ℵ₂ → (ℕ → 𝔹_cohen) :=
λ ν : ℵ₂, (λ n : ℕ, principal_open (ν,n))
```

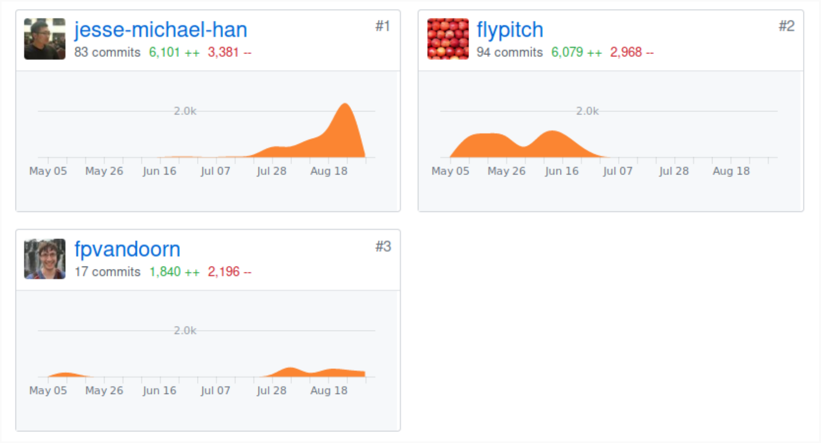
- `cohen_real.mk ν` is the Cohen real attached to ν, and it's straightforward to check that this induces an injection $\aleph_2 \hookrightarrow \mathcal{P}(\omega)$ in `bSet 𝔹` (so we have "added $\aleph_2$-many Cohen reals")
- Furthermore, one checks that $\mathbb{B}$ has the CCC, completing the forcing argument.

The final result

```
theorem CH_unprovable_from_ZFC : ¬ (ZFC′ ⊢′ CH_sentence) :=
begin
  intro H,
  suffices forces_false : ⊤ ⊩[V 𝔹] bd_falsum,
    from absurd (nontrivial.bot_lt_top) (not_lt_of_le
    forces_false),
  refine forced_absurd _ _, exact ZFC′, exact CH_f, swap, apply
    neg_CH_f,
  let prf_of_CH_f := sprovable_of_provable (classical.choice H),
  have CH_f_true := boolean_soundness prf_of_CH_f
    (V_𝔹_nonempty),
  convert CH_f_true, rw[inf_axioms_top_of_models
    (bSet_models_ZFC′ _)]
end
```

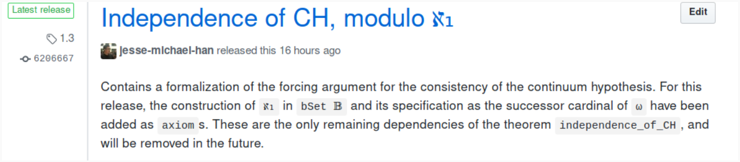
But wait, there's more

That was all done before April. Over the summer, we've added quite a bit more to the codebase. . .



But wait, there's more

- There's also a forcing argument for CH!
- It actually relies on more set theory (e.g. construction of $\aleph_1$, not just $\check{\aleph}_1$) than the Cohen forcing argument for $\neg$CH.
- Except for the (very) standard fact that ZFC proves the existence of a successor cardinal to ω, we have...



Independence of CH, modulo ℵ₁

jesse-michael-han released this 16 hours ago

Contains a formalization of the forcing argument for the consistency of the continuum hypothesis. For this release, the construction of `ℵ₁` in `bSet 𝔹` and its specification as the successor cardinal of `ω` have been added as `axiom` s. These are the only remaining dependencies of the theorem `independence_of_CH`, and will be removed in the future.

Forcing CH

- Whereas Cohen forcing creates a new injection $\aleph_2 \hookrightarrow \mathcal{P}(\omega)$, we can use collapse forcing to create a new surjection $\aleph_1 \twoheadrightarrow \mathcal{P}(\omega)$.

- Let $\mathbb{P}$`_collapse` be the poset of countable partial functions $\aleph_1 \to \mathcal{P}(\omega)$. The "principal open" sets

$$D_p := \{g : \aleph_1 \to \mathcal{P}(\omega) \mid g \text{ extends } p\}, \quad p \in \mathbb{P}_{\text{collapse}}$$

  form the basis of a topology τ on the function set $\mathcal{P}(\omega)^{\aleph_1}$. Put $\mathbb{B}$`_collapse := ` $\mathrm{RO}\left(\mathcal{P}(\omega)^{\aleph_1}, \tau\right)$.

Forcing CH

- To specify the surjection $\aleph_1 \twoheadrightarrow \mathcal{P}(\omega)$, need to specify a subset of the powerset $\mathcal{P}(\aleph_1 \times \mathcal{P}(\omega))$.
- Let $p(\nu, S)$ be the (singleton) countable partial function which sends $\nu : \aleph_1$ to $S : \mathcal{P}(\omega)$.
- Then we can check that the indicator function

$$(\nu, S) \mapsto D_{p(\nu,S)},$$

induces a surjection in `bSet` $\mathbb{B}$`_collapse` from $\check{\aleph}_1$ to $\mathcal{P}(\check{\omega})$.

- Then it remains to check that $\check{\aleph}_1 = \aleph_1$ and $\mathcal{P}(\omega) = \mathcal{P}(\check{\omega})$

Forcing CH

- This follows from the following property: for any `y : pSet`, and for any `f : bSet 𝔹_collapse`, if `bSet 𝔹_collapse` thinks that $f : \check{\omega} \to \check{y}$ is a function, then there exists a $g : \omega \to y$ in `pSet` such that $\check{g} = \check{f}$.

As before, reflect a boolean-valued ∀∃ statement into the metatheory. The following lemma is always true:

```
lemma AE_of_check_func_check (x y : pSet.{u}) {f : bSet 𝔹} {Γ : 𝔹}
  (H : Γ ≤ is_func' (x̌) (y̌) f) (H_nonzero : ⊥ < Γ) :
  ∀ (i : x.type),
   ∃ (j : y.type ) (Γ' : 𝔹) (H_nonzero' : ⊥ < Γ') (H_le : Γ' ≤ Γ),
    Γ' ≤ (is_func' (x̌) (y̌) f) ∧ Γ' ≤ (pair ((x.func i)ˇ ) ((y.func j)ˇ
   )) ∈ᴮ f :=
```

Forcing CH

Recursively applying this lemma, obtain values $g_0, \dots, g_n, \dots$ such that

$$\bot < \cdots < \left(\prod_{k \leqslant n} ((k, g_k) \in^{\mathbb{B}} g) \right) < \cdots < ((0, g_0) \in^{\mathbb{B}} g)$$

The intersection of this chain thinks that g is the required lift of f. In general, this intersection might be empty, but it is nonempty for $\mathbb{B}$`_collapse`, (essentially) because the union of a chain of countable partial functions is again a countable partial function. By a denseness argument, g has the required property.

Putting it all together, we have...

The final result

```
theorem CH_unprovable_from_ZFC : ¬ (ZFC′ ⊢′ CH_sentence) :=
unprovable_of_model_neg (V 𝔹_cohen) fundamental_theorem_of_forcing
  (nontrivial.bot_lt_top) V_𝔹_cohen_models_neg_CH

theorem neg_CH_unprovable_from_ZFC : ¬ (ZFC′ ⊢′ ∼CH_sentence) :=
unprovable_of_model_neg (V 𝔹_collapse) fundamental_theorem_of_forcing
  (nontrivial.bot_lt_top) (by {rw forced_in_not, from
    V_𝔹_collapse_models_CH})

def independent {L : Language} (T : Theory L) (f : sentence L) : Prop :=
¬ (T ⊢′ f ∨ T ⊢′ ∼f)

theorem independence_of_CH : independent ZFC′ CH_sentence :=
begin
  have := CH_unprovable_from_ZFC,
  have := neg_CH_unprovable_from_ZFC,
  finish
end
```

Automation for boolean-valued logic

- The calculus of the forcing relation (in the poset P of forcing conditions) can be thought of as a shorthand for the calculation of inequalities of boolean truth-values after embedding P into a boolean completion, i.e. $p \Vdash q \iff \widehat{p} \leqslant \widehat{q}$
- e.g., given a predicate $\phi(x)$, the sentence $\forall x, \phi(x)$ is "true" if we have that $\top \leqslant \prod_{x} \phi(x)$. Think of $\leqslant$ as a turnstile ($\vdash$).
- Calculations are hard if you manipulate these expressions as infinite sums and products.
- Calculations are easy if you think of these as formulas and replay the proofs from first-order logic.

Automation for boolean-valued logic

- In Lean, proving $p_1, ., p_k, .., p_n \vdash p_k$ is trivial (`by assumption`).
- Calculating $a_1 \sqcap \cdots \sqcap a_k \sqcap \cdots \sqcap a_n \leqslant a_k$ in $\mathbb{B}$ is also trivial, but harder to write a tactic that proves this uniformly in k and n.
- Solution: use the tactic framework to automate calculations in $\mathbb{B}$.
- Yoneda lemma: $a \leqslant b \iff \forall \Gamma, \Gamma \leqslant a \to \Gamma \leqslant b$.
- Applying this turns the previous problem into $(\Gamma : \mathbb{B}), \Gamma \leqslant a_1, \dots, \Gamma \leqslant a_k, \dots, \Gamma \leqslant a_n \vdash \Gamma \leqslant a_k$, which is now trivial (`by assumption`). Easy to automate.
- With more custom tactics/trickery (e.g. boolean-valued natural deduction tactics, coercing material implications and $\bigsqcap$s to Pi-types) we can pretend we're just writing FOL proofs.

Automation for boolean-valued logic

Old and busted:

```
example { 𝔹 } [complete_boolean_algebra 𝔹] {a b c : 𝔹} :
 ( a ⟹ b ) ⊓ ( b ⟹ c ) ≤ a ⟹ c :=
begin
  rw[<-deduction], unfold imp, rw[inf_sup_right, inf_sup_right],
  simp only [inf_assoc, sup_assoc], refine sup_le _ _,
  ac_change' (-a ⊓ a) ⊓ (-b ⊔ c) ≤ c,
  from inf_le_left_of_le (by simp), rw[inf_sup_right],
  let x := _, let y := _, change b ⊓ (x ⊔ y) ≤ _,
  rw[inf_sup_left], apply sup_le,
  { simp[x, inf_assoc.symm] },
  { from inf_le_right_of_le (by simp) }
end
```

Automation for boolean-valued logic

New hotness:

```
example { 𝔹 } [complete_boolean_algebra 𝔹] {a b c : 𝔹} :
 ( a ⟹ b ) ⊓ ( b ⟹ c ) ≤ a ⟹ c :=
by {tidy_context, bv_imp_intro, from a_1_right (a_1_left H)}

-- tactic state before final step:
-- a b c G : 𝔹,
-- G_1 : 𝔹 := a ⊓ G,
-- a_1_left : G_1 ≤ a ⟹ b,
-- a_1_right : G_1 ≤ b ⟹ c,
-- H : G_1 ≤ a
-- ⊢ G_1 ≤ c
```

Automation for boolean-valued logic

- This technique also exposes a family of setoids on `bSet` $\mathbb{B}$ induced by $\mathbb{B}$-valued equality: for every Γ, $\lambda x \quad y, \Gamma \leqslant x =^{\mathbb{B}} y$ is an equivalence relation.
- If the remainder of a proof is just equality reasoning (mod $\mathbb{B}$), we can just quotient by the setoid and run congruence closure.

```
example {a b c d e : bSet 𝔹} :
  (a =ᴮ b) ⊓ (b =ᴮ c) ⊓ (c =ᴮ d) ⊓ (d =ᴮ e) ≤ a =ᴮ e :=
by tidy_context; bv_cc
```

Proof transfer via completeness and boolean-valued soundness

- The $\mathbb{B}$-valued soundness theorem says that for any first-order sentence φ, any proof tree

$$\mathsf{ZFC} \vdash \varphi$$

  can be replayed in any $\mathbb{B}$-valued model.

- The completeness theorem says that if φ can be proven in every (ordinary, i.e. bool-valued) model of ZFC, then there is a proof tree $\mathsf{ZFC} \vdash \varphi$.

- While custom automation makes $\mathbb{B}$-valued proofs easier, we could also prove things like "Zorn's lemma is equivalent to AC" by working in an arbitrary ordinary model of ZFC, then transfer the proof to all $\mathbb{B}$-valued models of ZFC.

Formula pretty-printing

- Our FOL formulas use de Bruijn indices, which are not so fun to read.
- Solution: write a `print_formula` program in Lean to pretty-print them using named variables!

Example:

```
def axiom_of_regularity : sentence L_ZFC′ :=
  ∀′ (∼(&0 ≃ ∅′) ⟹ (∃′ (&′0 ∈′ &′1 ⊓ ∀′ (&′0 ∈′ &′2 ⟹ ∼(&′0 ∈′
    &′1)))))

#eval print_formula axiom_of_regularity
-- (∀x1,((x1 = ∅) ∨ ∃x2,(x2 ∈ x1∧ (∀x3,(x3 ∈ x1 ⟹ ¬x3 ∈ x2)))))
```

Formula parsing

- Lean also uses de Bruijn indices to represent variables, but parses them from a named representation entered by the user.
- So, use metaprogramming to hijack Lean's parser and pattern-match on the parsed `expr` to produce a deeply-embedded FOL formula!

Example:

```
def my_formula : formula L :=
by parse_formula (∀ x y : α, (x = y))

#print my_formula
-- ∀'∀'(&1 ≃ &0)
```

- (maybe) roll our own formula parser with a monadic parser combinator library

Summary

- Started in October 2018, now at 20,000 lines of code
- Our translation of the forcing argument into type theory shows that a ground model of set theory is not really needed to do forcing.
- Challenges: many parts of textbook expositions did not have type-theoretic analogues, and the forcing argument for CH via Boolean-valued models is not well-documented.
- Transfinite induction almost never needed once we have `bSet` $\mathbb{B}$ and its recursion principle.
- Domain specific automation is useful; Lean makes it easy to write.

Summary

Thank you!

- `flypitch.github.io`
- `https://www.github.com/flypitch/flypitch`