Towards a formal proof of the independence of the continuum hypothesis

Jesse Han

(joint with Floris van Doorn)

University of Pittsburgh

Lean Together 2019

The Flypitch project

The path to a proof

What's done

The way forward

What is CH?

- The continuum hypothesis (CH) states that there are no sets whose size is strictly larger than the countable natural numbers and strictly smaller than the uncountable real numbers, i.e.

$$\forall X, |X| > \aleph_0 \implies |X| \geqslant 2^{\aleph_0}.$$

- It was introduced by Cantor in 1878 and was the very first problem on Hilbert's list of twenty-three outstanding problems in mathematics.

What is CH?

- Gödel proved in 1938 that CH was consistent with ZFC (i.e. not disprovable from ZFC), and later conjectured that CH was independent of ZFC, i.e. neither provable nor disprovable from the ZFC axioms.
- In 1963, Paul Cohen developed *forcing*, which allowed him to prove the consistency of $\neg$CH with ZFC, and therefore complete the independence proof. For this work, he was awarded a Fields medal—the only one to ever be awarded for a work in mathematical logic.

What is the Flypitch project?

The Flypitch project aims to produce a formal proof of the independence of CH.

The Flypitch project

The path to a proof

What's done

The way forward

The path to a proof

The proof can be broken down into three steps:

1. Gödel's completeness theorem reduces "T does not prove ψ" to "there exists a model $M \models T$ such that $\neg\psi$ in M", and therefore the reduces problem to finding a model of ZFC where CH is true and a model of ZFC where CH is false.
2. Gödel's 1938 result produces a model of ZFC where CH is true.
3. Cohen's forcing argument produces a model of ZFC where CH is false.

The path to a proof

Before we can even formally *state* the problem, we need to formalize:

- Syntax of FOL
- A proof system
- Semantics of FOL
- Axioms of ZFC.

Related existing formalizations

- Harrison's formalization of first-order logic and basic model theory in HOL Light; also in the *Handbook of Practical Logic and Automated Reasoning*
- Lawrence Paulson's formalization of consistency of the axiom of choice (in Isabelle/ZF)
- Margetson, also Schlichtkrull: formalizations of first-order logic for verified provers (in Isabelle/HOL)
- Russell O'Connor's formal proof of Gödel incompleteness theorems (in Coq)
- Gunther et al: formalization of some axiomatic forcing (in Isabelle/ZF)

The Flypitch project

The path to a proof

What's done

The way forward

Overview

Code located at:
`https://github.com/flypitch/flypitch`.

- Work on current codebase started in October 2018.
- Now at $>$ 7500 lines of code.
- Finished: formalization of first-order logic and Gödel's completeness theorem.
- Also: compactness theorem, Peano arithmetic, Presburger arithmetic, ZFC.

Informal but detailed notes written with Lean's type-theoretic foundations, located at
`https://github.com/flypitch/flypitch-notes`.

Desiderata for a deep embedding of FOL

Syntax:

- Proof system should mirror `Prop`: natural deduction, intro/elim rules.
- Reflection-friendly: should be able to reify terms, formulas, proofs by structural induction; soundness should just match all deeply-embedded logical operations to their `Prop` counterparts.

Semantics:

- Given a model M, any term with k free variables should be interpreted as a k-ary function $M^k \to M$.
- Given a model M, any formula with k free variables should be interpreted as a k-ary function $M^k \to$ `Prop`

Languages

Function and relation symbols are types indexed by arity:

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

variable L : Language
```

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

def L_abel : Language := ⟨abel_functions, λn, empty⟩
```

Terms

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

@[reducible] 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.

Formulas

Formulas are defined analogously:

```
inductive preformula : ℕ → Type u
| falsum {} : preformula 0
| equal (t₁ t₂ : term L) : preformula 0
| 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
| all (f : preformula 0) : preformula 0

@[reducible] def formula := preformula L 0
```

A `preterm` (resp. `preformula`) of level `l` induces an `l`-ary function from terms to terms (resp. terms to formulas).

de Bruijn variables

> *Get used to using de Bruijn indices. (They are really not that bad.)*
>
> —*Jeremy Avigad*

- One of the reasons why the above syntax is so nice is because quantification is handled with de Bruijn variables: in a formula, the variable `&k` is bound by the kth outmost quantifier which has `&k` in scope.
- Variable shadowing does not happen with de Bruijn variables.
- Example: $\forall x, \forall y, x = y \implies (\forall z, z = x \implies z = y)$ becomes

$$\forall' \ \forall' \ \texttt{\&1} \simeq \texttt{\&0} \implies \forall' \ (\texttt{\&0} \simeq \texttt{\&2} \implies \texttt{\&0} \simeq \texttt{\&1})$$

Proof system

```
inductive prf : set (formula L) → formula L → Type u
| axm     {Γ A} (h : A ∈ Γ) : prf Γ A
| impI    {Γ : set (formula L)} {A B} (h : prf (insert A Γ
    ) B) : prf Γ (A ⟹ B)
| impE    {Γ} (A) {B} (h₁ : prf Γ (A ⟹ B)) (h₂ : prf Γ
    A) : prf Γ B
| falsumE {Γ : set (formula L)} {A} (h : prf (insert ∼A Γ
    ) ⊥) : prf Γ A
| allI    {Γ A} (h : prf (lift_formula1 ″ Γ) A) : prf Γ
    (∀′ A)
| allE₂   {Γ} A t (h : prf Γ (∀′ A)) : prf Γ (A[t // 0])
| ref     (Γ t) : prf Γ (t ≃ t)
| subst₂  {Γ} (s t f) (h₁ : prf Γ (s ≃ t)) (h₂ : prf Γ
    (f[s // 0])) : prf Γ (f[t // 0])
```

Sentences

To define sentences, need a way to rule out free variables. Since we use de Bruijn variables, suffices to ensure index of largest variable never exceeds quantifier depth. To make it easy to structurally induct, we define `bounded_preterm` and `bounded_preformula`:

```
inductive bounded_preterm (n : ℕ) : ℕ → Type u
| bd_var {} : ∀ (k : fin n), bounded_preterm 0
| bd_func {} : ∀ {l : ℕ} (f : L.functions l),
    bounded_preterm l
| bd_app : ∀ {l : ℕ} (t : bounded_preterm (l + 1))
    (s : bounded_preterm 0), bounded_preterm l

def bounded_term   (n) := bounded_preterm L n 0
def closed_preterm (l) := bounded_preterm L 0 l
def closed_term        := closed_preterm L 0
```

Sentences

```
inductive bounded_preformula : ℕ → ℕ → Type u
| bd_falsum {} {n} : bounded_preformula n 0
| bd_equal {n} (t₁ t₂ : bounded_term L n) :
    bounded_preformula n 0
| bd_rel {n l : ℕ} (R : L.relations l) :
    bounded_preformula n l
| bd_apprel {n l} (f : bounded_preformula n (l + 1))
    (t : bounded_term L n) : bounded_preformula n l
| bd_imp {n} (f₁ f₂ : bounded_preformula n 0) :
    bounded_preformula n 0
| bd_all {n} (f : bounded_preformula (n+1) 0) :
    bounded_preformula n 0

def sentence : bounded_preformula 0 0
```

Semantics

Without a bound, cannot instantiate a formula without knowing what to assign to *every* `&k`, $k : \mathbb{N}$.
With a bound, we can use vectors of elements instead.

```
inductive dvector (α : Type u) : ℕ → Type u
| nil {} : dvector 0
| cons : ∀{n} (x : α) (xs : dvector n), dvector
    (n+1)
```

Semantics

```
@[simp] def realize_bounded_term {S : Structure L}
    {n} (v : dvector S n) : ∀{l} (t :
    bounded_preterm L n l) (xs : dvector S l),
    S.carrier
| _ &k              xs := v.nth k.1 k.2
| _ (bd_func f)     xs := S.fun_map f xs
| _ (bd_app t₁ t₂) xs := realize_bounded_term t₁
    (realize_bounded_term t₂ ([])::xs)
```

Semantics

```
@[simp] def realize_bounded_formula :
  ∀{n l} (v : dvector S n) (f : bounded_preformula L
    n l) (xs : dvector S l), Prop
| _ _ v bd_falsum xs := false
| _ _ v (t₁ ≃ t₂) xs := realize_bounded_term v t₁ xs
    = realize_bounded_term v t₂ xs
| _ _ v (bd_rel R) xs := S.rel_map R xs
| _ _ v (bd_apprel f t) xs :=
    realize_bounded_formula v f
    (realize_bounded_term v t ([])::xs)
| _ _ v (f₁ ⟹ f₂) xs := realize_bounded_formula v
    f₁ xs → realize_bounded_formula v f₂ xs
| _ _ v (∀' f) xs := ∀(x : S),
    realize_bounded_formula (x::v) f xs
```

Completeness: statement and proof sketch

Let T be a theory and ψ a sentence. Gödel's completeness theorem states that the following are equivalent:

1. T proves ψ.
2. ψ is satisfied in every model of T.

Proof sketch:

1. $1 \Rightarrow 2$ is the soundness theorem, which follows from a structural induction on proofs.
2. Conversely, if $\neg(T \vdash \psi)$, then $T \cup \neg\psi$ is consistent. Any model of this is a model of T. Obtain a model by the Henkin construction which satisfies $\neg\psi$, which gives the contrapositive of $2 \Rightarrow 1$.

Completions of theories

- The Henkin argument requires completions of theories.
- Applied Johannes' implementation of Zorn's lemma.
- Could have interpreted theories as filters on the poset of formulas, but needed to define filter machinery for posets instead of just `set (set` α`)`
- To apply Zorn's lemma, needed to bundle and work with the poset structure on the subtype of consistent extensions of a fixed theory T.

Term models

- The Henkin construction produces a model of T using terms of the language (provided the language has enough constants to witness all existential statements proved by T).
- First serious test of both syntax and semantics
- Proof hinges on proving completeness for term models of complete Henkinized theories and then reducing to this case
- This is an example of where straightforward structural induction doesn't work; needed to induct on quantifier depth instead and then structurally induct on quantifier-free sentences.

Language morphisms

Henkinization requires the construction of an infinite chain of languages and taking the union. Since we're working with types, we need to work with injective maps between Languages instead:

$$\mathcal{L} = \mathcal{L}_0 \to \mathcal{L}_1 \to \cdots \to \mathcal{L}_n \to \ldots; \mathcal{L}_\infty = \varinjlim \mathcal{L}_i$$

```
structure Lhom (L L′ : Language) :=
  (on_function : ∀{n}, L.functions n → L′.functions
    n)
  (on_relation : ∀{n}, L.relations n → L′.relations
    n)
```

Lifting proofs along language morphisms

A nontrivial and crucial but overlooked (in nearly every source we found) step in the Henkin argument: let $\mathcal{L}_1 \to \mathcal{L}_2$ be an injective language morphism. Let Γ be a set of $\mathcal{L}_1$-formulas and let f be an $\mathcal{L}_1$-formula. If there is a proof $\Gamma \vdash \psi$ in the language $\mathcal{L}_2$, there exists a proof $\Gamma \vdash \psi$ in the original language $\mathcal{L}_1$.

```
/- here F is an Lhom, F : L → L' -/
noncomputable def reflect_prf {Γ : set (formula L)}
    {f : formula L} (hF : F.is_injective)
  (h : F.on_formula '' Γ ⊢ F.on_formula f)
   : Γ ⊢ f
```

Colimits and Henkinization

To perform the Henkin construction, need to take colimits of languages. We rolled our own (filtered) colimits (`src/colimit.lean`).
To show "obvious" (but painful to prove) things like "the Henkinization of T is a Henkin theory", need to:

- Construct the induced colimits on preterms, terms, preformulas, formulas
- Show the universal comparison maps are bijective (e.g. $\varinjlim$ `preterm L_n l` $\simeq$ `preterm L_infty l`)
- Take each `T_n : Theory L_n`, push it into `Theory L_infty`, and work with the resulting (literal) chain of sets (e.g. showing it's a chain).

Putting it all together

```
theorem model_existence {L : Language} (T : Theory
    L) : is_consistent T ↔ (∃ M : Structure L,
    (nonempty M) ∧ M ⊨ T)

theorem completeness {L : Language} (T : Theory L)
    (ψ : sentence L) : T ⊢' ψ ↔ T ⊨ ψ

theorem compactness {L : Language} {T : Theory L} {f
    : sentence L} :
  T ⊨ f ↔ ∃ fs : finset (sentence L), (↑fs :
    Theory L) ⊨ (f : sentence L) ∧ ↑fs ⊆ T
```

Pros and cons of our implementation

Pros:

- Well-formedness *is* type-correctness
- Many proofs become straightforward structural inductions easily handled with the equation compiler or `induction on x; simp*`
- Can write examples of `bounded_formula` with the same notation as `formula` without requiring separate proofs of boundedness ("the notation *is* the proof of boundedness")
- Well-behaved semantics. Soundness is straightforward, as it should be. Statements like

```
ℤ' ⊨ ∀'∀'(&1 +' (&0 +' &0) ≃ &0 +' (&0 +' &1)) ↔
    ∀ (x y : ℤ), x + (y + y) = y + (y + x)
```

hold `by refl`.

Pros and cons of our implementation

Cons:

- We often need `casts` to ensure statements involving `bounded_terms` type-check, usually induced by $n \leqslant m$ or equalities like $n + 2 = 1 + n + 1$, e.g. "the realizations of a closed term at any vector of any length are all the same", or "substituting at the first open variable after taking forall is the same as taking forall after substituting at the second open variable":

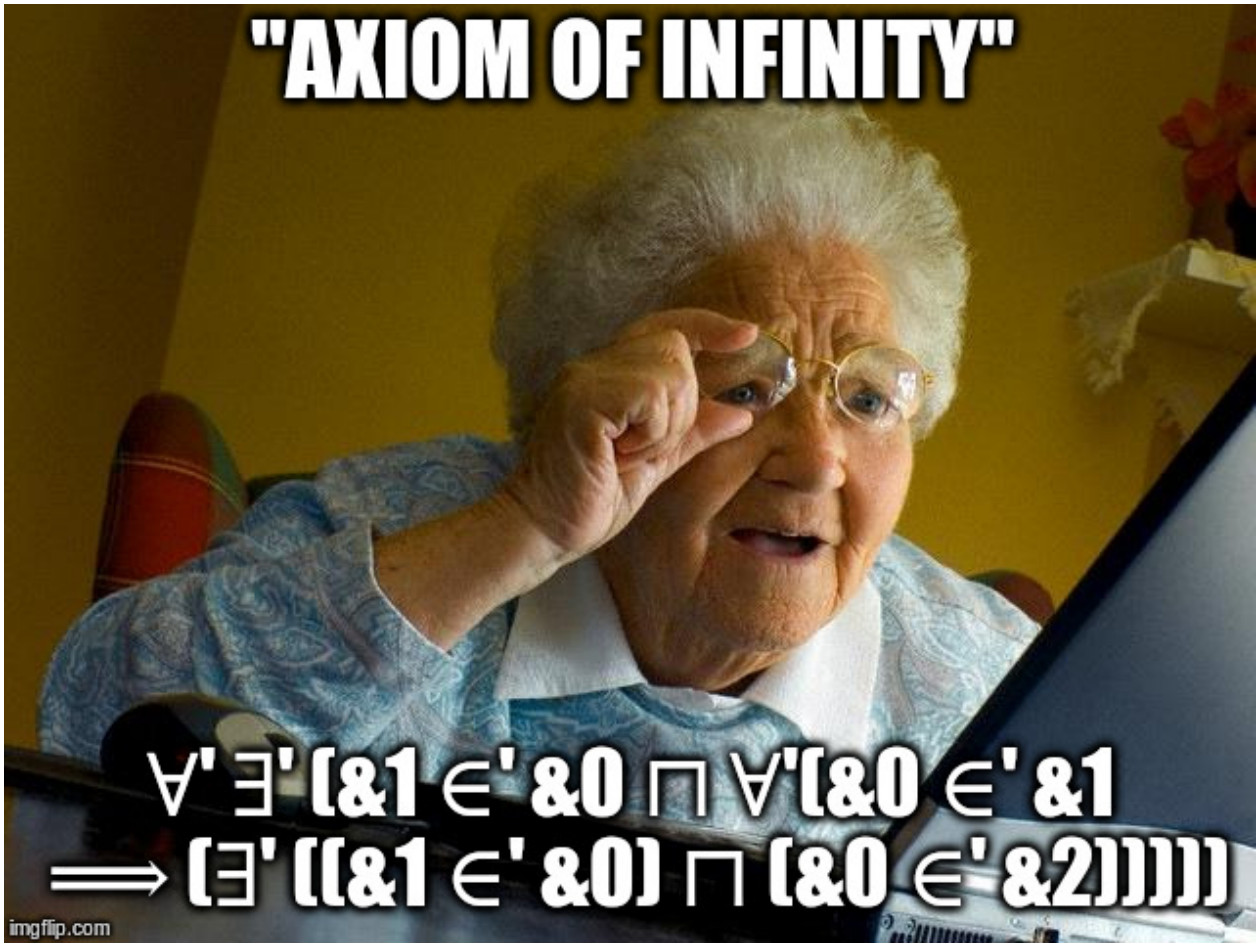
```
((∀'f)[t.cast (by simp) /0] : bounded_formula L
    n) = ∀'((f[t.cast (by {simp} : 0 ⩽ n) // 1
    // (by {simp} : 1 + n + 1 = n + 2)]).cast_eq
    (by simp))
```

- (This is actually a "`by tidy`" one-liner, but related statements are often not.)

Pros and cons of our implementation

Cons:

- de Bruijn variables are easy for computers to read, not so much for humans as quantifier depth increases and substitutions/lifts show up:



Peano arithmetic

Example of the sentence notation at work:

```
inductive peano_functions : ℕ → Type
| zero : peano_functions 0
| succ : peano_functions 1
| plus : peano_functions 2
| mult : peano_functions 2

def L_peano : Language :=
  ⟨peano_functions, λ n, empty⟩

def p_induction_schema {n : ℕ} (ψ : bounded_formula
    L_peano (n+1)) : sentence L_peano :=
bd_alls n (ψ[zero/0] ⊓ ∀' (ψ ⟹ (ψ ↑' 1 # 1)[succ
    &0/0]) ⟹ ∀' ψ)
```

ZFC

Can either define abbreviations like $\subseteq$ or $\{x, y\}$ externally (extensions of language) or internally. Current development is *bona fide* ZFC with just the symbol $\{\in\}$.

Over the weekend, Andrew Tindall completed the formal statement of ZFC:

```
def axiom_of_choice : sentence L_ZFC :=
∀'∀'(fn_domain ⟹
        ∃'∀'(&0 ∈' &2 ⟹
                ∀'(fn_app ↑' 1 # 2 ↑' 1 # 2 ⟹
                    (∀'(fn_app ↑' 1 # 1 ↑' 1 # 4 ↑' 1 # 4 ⊓ ∃
    '(&0 ∈' &2)) ⟹ &0 ∈' &1))))

def ZF : Theory L_ZFC := {axiom_of_extensionality,
    axiom_of_union, axiom_of_powerset, axiom_of_infinity}
    ∪ (λ(c : bounded_formula L_ZFC 2),
    axiom_of_replacement c) '' set.univ

def ZFC : Theory L_ZFC := ZF ∪ {axiom_of_choice}
```

Statement of CH

...and also a statement of CH!

```
def continuum_hypothesis : sentence L_ZFC := ∀'
    ∀'  ((∃'((is_first_infinite_ordinal ↑' 1 # 1 ↑'
     1 # 1) ⊓ (is_powerset ↑' 1 # 2)) ⊓
    (is_first_uncountable_ordinal ↑' 1 #0)) ⟹
    zfc_equiv)

-- TODO
theorem independence_of_CH :
  (¬ ZFC ⊢' continuum_hypothesis) ∧ (¬ ZFC ⊢'
    ~ continuum_hypothesis) := sorry
```

From a Math.SE thread about formally writing out the continuum hypothesis:

1 "long" as in a few pages or a few googols of pages? – djechlin Jan 29 '16 at 0:16

4 No. If you really try you can probably fit it in a single A4 page. Probably less (depending on your font size and margins, but let's assume standard `fullpage` LaTeX in 10pt TeX Gyre font). But not something I'd care to write explicitly on this site, and not something I think is *worth* being written explicitly on this site either. – Asaf Karagila ♦ Jan 29 '16 at 0:17

Formula pretty-printer

Disclaimer: code under construction!

The way forward

To-do list

(or: La longue marche á travers la thèorie des ensembles):

- Lowenheim-Skolem theorems
- show ZFC is consistent (by e.g. showing Mario's "model of ZFC" is a model of ZFC)
- Formalize the constructible universe L
- Extract a countable model for forcing
- Formalize axiomatic presentation of forcing (without reference to countable transitive models of set theory)

To-do list

Long march aside, other useful things would be:

- Custom tactics/automation for tedious work (like proving deeply-embedded tautologies or parsing de Bruijn indices).
- Verified reflection procedure for first-order typeclasses
- Extensive documentation (coming soon: `flypitch-doc`) to reduce activation energy for future contributions. Also integration with the Formal Abstracts project.

Wishlist

Maybe:

- Construct $\mathbb{Q}^{\text{alg}}$ as the term model of the Henkinization of ACF_0?
- Go another level down and embed first-order logic inside a model of ZFC?
- Categorical logic and sheaf-theoretic forcing?

Final remarks

- Contributions and PRs are always welcome (see open issues at Github page)
- de Bruijn indices really aren't that bad
- Lean is fun!
- A big thank you to Floris for being an excellent mentor!

Thank you!