Flypitch Lean 4

zj

April 12, 2026

Contents

Chapter 1

Overview

The long-term goal of Flypitch is a formal proof that the continuum hypothesis is independent of ZFC. Mathematically, the original project reaches this goal by combining three ingredients:

- a first-order logic and completeness development,
- the forcing and Boolean-valued-model machinery,
- the set-theoretic infrastructure connecting these two sides.

The present Lean 4 repository does not yet cover this entire story. What is already formalized is the logic-side development leading from first-order syntax to complete Henkin extensions of consistent theories. This blueprint is therefore written as an account of the mathematics that is currently available, rather than as a premature sketch of the final independence proof.

1.1 What Has Been Built

The existing Lean 4 development already contains a coherent model-theoretic thread. Starting from the syntax and semantics of first-order logic, it proves soundness, establishes compactness in the proof-theoretic form needed for later consistency arguments, constructs maximal consistent extensions, and develops the language-extension machinery required for Henkinization. The endpoint of the present port is a complete Henkin extension of any consistent theory.

This means that the current repository already supports a substantial part of the standard completeness argument. What is still missing is the later packaging of completeness itself and, beyond that, the forcing and set-theoretic constructions that are needed for the independence result.

1.2 How To Read The Blueprint

This blueprint is organized as a mathematical progression.

- Chapter 2 develops first-order languages, syntax, derivability, semantics, and theories.
- Chapter 3 explains how consistency can be reduced to finite fragments and how consistent theories are enlarged to complete ones.

- Chapter 4 introduces the language maps, reducts, colimits, and reflection arguments needed to compare syntax across enlarging languages.
- Chapter 5 uses these ingredients to build the Henkin language, the limit theory, and finally a complete Henkin extension.
- Chapter 6 records the current boundary of the Lean 4 port.

The intended reading is therefore not "which file was written first", but rather "which mathematical idea depends on which earlier one".

1.3 The Current Frontier

The main point of this blueprint is that the frontier of the Lean 4 port no longer lies in the elementary first-order material. The logic-side development has already progressed through the Henkin construction. The next serious gap is after this stage: first the remaining completeness-side packaging, and then the forcing branch beginning with the still-unported `pSet_ordinal` development.

1.4 Formalization Note

The source files behind this blueprint include the first-order logic modules, the compactness and completion files, the colimit and language-extension files, and the Henkin file. Their role in the present document is supportive: the chapters below describe the mathematical constructions first, and only then indicate where those constructions are realized in Lean.

Chapter 2

First-Order Logic Core

The first task is to fix a precise first-order language, define its syntax and semantics, and specify what it means to prove a formula from hypotheses. This chapter records the formalized core of that theory. It corresponds to the Lean files `Flypitch/FOL/Syntax.lean`, `Flypitch/FOL/Formula.lean`, `Flypitch/FOL/Proof.lean`, `Flypitch/FOL/Semantics.lean`, `Flypitch/FOL/Theory` and `Flypitch/FOL/Bounded.lean`.

2.1 Languages, Terms, And Formulas

Definition 1 (First-Order Language)**.** A first-order language consists of function symbols and relation symbols, each sorted by arity.

Once a language L is fixed, one forms its terms and formulas in the usual way. The implementation uses de Bruijn variables, so bound variables are tracked numerically rather than by names. This choice is mathematically inessential, but it makes substitution and variable-shifting precise enough for the later completeness arguments.

The syntax is packaged in a slightly unusual arity-sensitive form: terms and formulas are first built as partially applied expressions and only later specialized to the closed cases. The advantage is that the basic structural operations can be treated uniformly across all arities.

Definition 2 (Structural Operations)**.** The syntax carries two fundamental operations:

- lifting, which raises free variables above a chosen cutoff;
- substitution, which replaces a free variable by a term.

These are the bookkeeping operations behind every later argument. They are needed to formulate quantifier rules, to compare syntax under language maps, and to express the witness constructions appearing in the Henkin chapter.

2.2 Derivability

Definition 3 (Derivability)**.** For a set Γ of formulas, the relation $\Gamma \vdash A$ is defined by a natural-deduction proof system with rules for assumptions, implication, universal quantification, falsity, equality, and substitution of equals.

This is the proof-theoretic core of the development. The formalized system is strong enough for the compactness and completion arguments later on, and the repository proves the expected structural facts such as weakening and a family of derived introduction and elimination rules.

2.3 Structures And Satisfaction

Definition 4 (Structure)**.** An L-structure consists of a carrier set together with interpretations of all function and relation symbols of L.

Given a structure M and a valuation of variables in M, every term acquires an interpretation in M and every formula acquires a truth value. Closed formulas may therefore be discussed without reference to a valuation, and one obtains the usual notion of semantic consequence.

The central compatibility theorem in this part of the development identifies syntactic substitution with semantic reassignment of variables. This is what allows the proof system to interact correctly with semantics.

Proposition 5 (Soundness)**.** *If $\Gamma \vdash A$, then every structure satisfying all formulas in Γ also satisfies A.*

$$\Gamma \vdash A \implies \Gamma \vDash A.$$

Proof. The proof is by induction on a derivation. Each inference rule is checked directly against the semantic definition of truth, with the lifting and substitution lemmas handling the quantifier and equality cases. □

2.4 Sentences, Theories, And Bounded Syntax

The later chapters work primarily with closed formulas, that is, with sentences. A *theory* is therefore taken to be a set of sentences. At this level one can define theory-level provability and satisfaction, together with the familiar notions of consistency and completeness.

The development also isolates formulas with a prescribed number of free variables. This is not merely a technical refinement. The Henkin construction needs to speak uniformly about formulas in one free variable and then substitute closed terms into them.

Definition 6 (Bounded Syntax)**.** For each natural number n, a bounded term or formula is a term or formula whose free variables lie among the first n variables.

These bounded objects are the correct language for witness formulas, witness constants, and the quantified generalization arguments that appear later.

2.5 A Model-Theoretic Example

The file `Flypitch/Examples/Abel.lean` supplies a concrete example in the language of abelian groups. The axioms are interpreted in the integers, and the formalization verifies that the integer structure satisfies them.

Proposition 7 (Integers Satisfy The Abelian-Group Axioms)**.** *The standard structure on $\mathbb{Z}$ is a model of the chosen theory of abelian groups.*

This example has a clear mathematical purpose. It shows that the syntax, semantics, and proof system already interact correctly in a familiar setting before the blueprint moves on to the abstract compactness and completion machinery.

2.6 Formalization Note

In Lean, the basic objects are implemented as the structures and types `Language`, `preterm`, `preformula`, `prf`, `Structure`, `Theory`, `bounded_term`, and `bounded_formula`. These names are useful when reading the code, but the mathematical content of the chapter is the ordinary first-order framework described above.

Chapter 3

Compactness And Completion

The next step is to understand how consistency behaves under enlarging a theory. The formalized route is the classical one: first prove that any proof uses only finitely many assumptions, then use this finitary principle to build maximal and hence complete consistent extensions. This chapter corresponds to `Flypitch/Compactness.lean` and `Flypitch/Completion.lean`.

3.1 Compactness As A Finitary Principle

The later Henkin construction repeatedly adds new sentences to a theory. To show that these enlargements preserve consistency, it is enough to know that an inconsistency can already be witnessed inside some finite fragment. The repository proves exactly this statement in proof-theoretic form.

Theorem 8 (Proof Compactness)**.** *If a formula ψ is derivable from a set T of formulas, then there exists a finite subset $\Gamma \subseteq T$ such that $\Gamma \vdash \psi$.*

Proof. The proof is by induction on a derivation of ψ. Each inference rule uses only finitely many assumptions, and finite supports are combined by taking finite unions. The only delicate point is universal introduction, where one must descend from a lifted theory back to a finite family of original assumptions. □

This is the precise finiteness statement needed later: a contradiction cannot depend essentially on infinitely many hypotheses at once.

3.2 Compactness For Theories

The rest of the development is phrased in terms of theories of sentences rather than arbitrary sets of formulas, so the same result is repackaged at the theory level.

Theorem 9 (Theory Compactness)**.** *If a sentence ψ is provable from a theory T, then there is a finite subtheory $\Gamma \subseteq T$ from which ψ is already provable.*

This is the form used in later consistency arguments. Whenever a large theory is inconsistent, the obstruction already appears in a finite collection of its sentences.

3.3 Controlled Unions

Compactness immediately gives a useful principle for enlarging consistent theories.

Proposition 10 (Consistency Of A Controlled Union)**.** *Let T_1 be a consistent theory. Suppose that every sentence ψ in a second theory T_2 can be added to T_1 without forcing inconsistency in the relevant one-step sense. Then the union $T_1 \cup T_2$ is consistent.*

Proof. Assume the union were inconsistent. By Theorem 9, some finite fragment is already inconsistent. One then adds the finitely many sentences coming from T_2 one at a time and uses the hypothesis at each step to rule out the appearance of a contradiction. □

This proposition is the first general tool showing that a large extension can be handled by checking one sentence at a time. It will be reused in the Henkin chapter.

3.4 Chains Of Consistent Extensions

To pass from a consistent theory to a maximal one, one considers the partially ordered set of all consistent extensions ordered by inclusion. The key point is that any chain of such extensions has an upper bound, namely its union.

Lemma 11 (Finite Fragments Lie In One Chain Element)**.** *Let c be a nonempty chain of consistent extensions of a theory T. Every finite subset of the union of the chain is already contained in a single member of the chain.*

This is immediate from finite induction and total comparability inside the chain.

Proposition 12 (Consistency Of The Union Of A Chain)**.** *The union of a chain of consistent extensions of T is again consistent.*

Proof. If the union were inconsistent, compactness would produce a finite inconsistent subtheory. By Lemma 11, that finite fragment would already lie in one element of the chain, contradicting the consistency of that element. □

3.5 Maximal Consistent Extensions

Once unions of chains are known to preserve consistency, Zorn's lemma applies.

Theorem 13 (Maximal Consistent Extension)**.** *Every consistent theory has a maximal consistent extension.*

Maximality then yields the expected dichotomy for sentences.

Proposition 14 (A Maximal Consistent Extension Is Complete)**.** *If $T_{\max}$ is maximal among the consistent extensions of a theory T, then $T_{\max}$ is complete.*

Proof. Let ψ be any sentence. If neither ψ nor $\neg\psi$ lies in $T_{\max}$, then one of the two one-sentence enlargements remains consistent. That contradicts maximality. □

Corollary 15 (Completion Of A Consistent Theory)**.** *Every consistent theory has a complete consistent extension.*

3.6 Why This Matters Later

The role of this chapter is straightforward.

- Compactness turns inconsistency questions into finite calculations.
- Controlled unions show how to adjoin families of new sentences while tracking consistency.
- Completion turns a merely consistent theory into one that decides every sentence.

These are exactly the abstract ingredients needed before witness constants can be adjoined systematically in the Henkin construction.

3.7 Formalization Note

The Lean development packages these ideas through the theorems `proof_compactness`, `theory_proof_compactness`, `is_consistent_union`, `consis_limit`, `maximal_extension`, and `complete_maximal_extension_of_consis`. The mathematical content, however, is the standard compactness-and-Zorn argument just described.

Chapter 4

Colimits And Language Extensions

The Henkin construction enlarges the language repeatedly. To control this process one needs a way to compare formulas written in different languages, to restrict structures along language maps, and to form the direct limit of an infinite chain of languages. This chapter describes that infrastructure. It is implemented in `Flypitch/Colimit.lean` and `Flypitch/LanguageExtension.lean`.

4.1 Directed Systems And Their Colimits

Suppose one has a directed family of objects connected by compatible transition maps. The relevant colimit is obtained by taking the disjoint union of all stages and identifying two representatives when they eventually agree at some later common stage.

Definition 16 (Directed Diagram)**.** A directed diagram is a family of objects indexed by a directed preorder, together with compatible transition maps along the preorder.

Definition 17 (Colimit)**.** The colimit of a directed diagram is the quotient of the disjoint union of the stages by eventual equality.

This is the right construction for the infinite Henkin language: a symbol or formula appearing at some finite stage should be regarded as the same object as its image in every later stage.

Proposition 18 (Canonical Maps And Universal Property)**.** *Each stage maps canonically into the colimit, and the colimit satisfies the expected universal property with respect to compatible families of maps out of the diagram.*

When the transition maps are injective, the canonical maps into the colimit are injective as well. This allows the finite stages to be viewed as genuine substructures of the limit object.

4.2 Maps Of Languages

Definition 19 (Language Homomorphism)**.** A homomorphism of first-order languages $L \to L'$ is an arity-preserving map on function symbols together with an arity-preserving map on relation symbols.

Such a map sends every term and formula over L to a corresponding term or formula over L'. In effect, one may reinterpret a proof written in the smaller language inside the larger one.

4.3 Symbol Support And Filtered Languages

Later witness arguments require careful control over which constants actually occur in a term or formula. For that reason the development records the set of symbols appearing in a given syntactic object and studies how this set behaves under lifting, substitution, and transport along language maps.

Definition 20 (Filtered Language)**.** Given a predicate on the symbols of a language L, the filtered language keeps exactly the symbols satisfying that predicate.

The key point is reconstruction: if every symbol appearing in a term or formula satisfies the predicate, then that term or formula already comes from the filtered language. This is the mechanism later used to remove a forbidden constant from the ambient language.

4.4 Proof Transport And Reducts

Language maps act on both syntax and semantics.

Proposition 21 (Proof Transport)**.** *A derivation in a language L can be transported along a language map $L \to L'$ to a derivation in the larger language L'.*

In the opposite direction, an L'-structure can be restricted to an L-structure by forgetting the interpretations of symbols not coming from L.

Definition 22 (Reduct)**.** Given a language map $\varphi : L \to L'$ and an L'-structure M, the reduct of M along φ is the L-structure obtained by restricting the interpretation to the image of L.

This restriction behaves exactly as expected: a transported formula is true in M if and only if the original formula is true in the reduct.

4.5 Reflection Along Injective Language Maps

Transporting syntax forward is easy. Going backward is subtler, because symbols in the larger language need not come from the smaller one. The formalized solution is a controlled reflection procedure.

Definition 23 (Reflection)**.** For an injective language map with decidable image, reflection attempts to pull terms and formulas in the larger language back to the smaller language. Symbols outside the image are replaced by variables.

The mathematical point is that genuinely new constants are not ignored; they are turned into open slots. This makes it possible to convert a proof involving a fresh constant into a proof of a universally quantified statement.

At the proof-theoretic level, reflection shows that injective language extensions preserve consistency of the theories induced from the smaller language.

4.6 Fresh Constants And Generalization

Definition 24 (Substituting A Closed Term Into A One-Variable Formula)**.** If $f(x)$ is a bounded formula in one free variable and t is a closed term, then one may form the sentence $f(t)$ by substitution.

The main application of reflection is the following familiar principle.

Theorem 25 (Generalization From A Fresh Constant)**.** *Suppose a constant symbol c does not occur in the assumptions Γ and does not occur in the formula $f(x)$. If Γ proves the sentence $f(c)$, then Γ proves $\forall x\, f(x)$.*

Proof. One removes the forbidden constant from the language by passing to a filtered language in which c is absent. Since neither the assumptions nor the formula mention c, they can be reconstructed inside that smaller language. Reflection then turns the proof of $f(c)$ into a proof of the same formula with the constant replaced by a variable, which is exactly what is needed for universal generalization. □

This theorem is the technical heart of the later witness argument. It explains why a proof using a genuinely fresh constant may be converted into a proof of a universal statement with no reference to that constant.

4.7 Role In Henkinization

This chapter serves three purposes at once.

- Directed colimits provide the eventual infinite language.
- Language maps and reducts let syntax, proofs, and models move between different stages of that language.
- Reflection and fresh-constant generalization supply the key argument that one-step witness adjunction preserves consistency.

4.8 Formalization Note

The supporting Lean declarations include `colimit`, `canonical_map`, `Lhom`, `on_prf`, `reduct`, `reflect_formula`, and `generalize_constant`. They are best read as implementations of the mathematical constructions above, not as substitutes for them.

Chapter 5

Henkinization

We now reach the central construction on the logic side. Starting from a consistent theory, one adjoins enough witness constants to ensure that every existential statement has a named witness in an enlarged language. The output is a consistent Henkin theory, and after applying completion one obtains a complete Henkin extension. This chapter documents the part of that argument implemented in `Flypitch/Henkin.lean`.

5.1 Adjoining Witness Constants In One Step

Given a language L, the first move is to enlarge it by adding one new constant for each bounded formula in one free variable.

Definition 26 (One-Step Henkin Language)**.** The one-step Henkin extension of L is the language obtained by adjoining, for every bounded formula $f(x)$, a new constant intended to witness f.

This is the formal version of the usual idea from the completeness proof: if one wants existential statements to have canonical witnesses, one expands the language so that such witnesses can be named.

5.2 Iterating The Construction

One application of the previous step is not enough, because after adjoining new constants, one obtains new formulas that may themselves require witnesses. The construction is therefore iterated along the natural numbers.

Definition 27 (Henkin Language Chain)**.** The Henkin language chain starts from the original language at stage 0 and applies the one-step witness extension at each successor stage.

The transition maps between stages are injective, so each finite stage embeds faithfully into every later one.

5.3 The Infinite Henkin Language

Definition 28 (The Limit Language)**.** The infinite Henkin language L_∞ is the directed colimit of the finite Henkin language chain.

Mathematically, L_∞ is the language obtained by adjoining all witness constants produced at all finite stages and identifying symbols that are the same after passing sufficiently far along the chain.

5.4 Finite-Stage Representatives

The crucial structural fact about L_∞ is that terms and formulas over the limit language still come from finite stages.

Theorem 29 (Finite-Stage Comparison)**.** *Every term, formula, bounded term, and bounded formula over L_∞ is represented by a unique compatible germ of corresponding syntax at a finite stage of the Henkin chain.*

Proof. Injectivity comes from the injectivity of the canonical maps into the colimit. Surjectivity is proved by structural recursion: each finite syntactic piece of an L_∞-expression already uses only finitely many symbols, so all of them appear together at some sufficiently high finite stage. □

This theorem is what allows witness arguments in the limit language to be reduced to witness arguments at finite stages.

5.5 Witness Sentences And Enough Constants

Definition 30 (Witness Property)**.** For a formula $f(x)$ and a constant c, the witness property is the sentence

$$\exists x\, f(x) \to f(c).$$

A theory has *enough constants* if every bounded one-variable formula has some constant satisfying this witness property.

5.6 The One-Step Theory Extension

Definition 31 (One-Step Henkin Theory)**.** Given a theory T over L, the one-step Henkin extension is obtained by transporting T into the one-step Henkin language and adjoining all witness sentences for bounded one-variable formulas over L.

This is the natural theory-level companion to the language extension: each new constant is accompanied by the axiom asserting that it behaves as a witness.

5.7 Consistency Of One Henkin Step

Theorem 32 (Consistency Of One Henkin Step)**.** *If T is consistent, then its one-step Henkin extension is also consistent.*

Proof. The key point is that each newly adjoined witness constant is fresh. If a finite collection of witness axioms produced a contradiction, one could add them one at a time. At each stage, the fresh-constant generalization theorem from Chapter 4 converts any illicit proof using the new constant into a proof that no longer depends on it. Compactness reduces the argument to finitely many such steps. □

This is the decisive consistency-preservation result in the entire Henkin construction.

5.8 Passing To The Limit Theory

The one-step theorem is then iterated along the chain of finite Henkin stages, and each finite-stage theory is transported into the limit language L_∞. Their union is the limit theory usually denoted T_∞.

Definition 33 (The Limit Theory)**.** The theory T_∞ is the union, inside L_∞, of the images of all finite Henkin stages of the original theory.

Theorem 34 (Consistency Along The Henkin Chain)**.** *If T is consistent, then every finite Henkin stage is consistent, the corresponding theories inside L_∞ are consistent, and the limit theory T_∞ is consistent.*

Proof. Finite-stage consistency is an induction using Theorem 32. The consistency of the union again comes from compactness: any finite inconsistent fragment of T_∞ already lies in a single finite stage. □

5.9 Enough Constants In The Limit

The next point is to show that the limit theory really has the Henkin witness property. Given a bounded one-variable formula over L_∞, the comparison theorem above represents it at some finite stage. The corresponding witness sentence has already been inserted at the next stage, and therefore appears in the limit theory after transport to L_∞.

Proposition 35 (Henkinization Has Enough Constants)**.** *The consistent limit theory obtained from the Henkin chain has enough constants.*

5.10 Complete Henkin Extensions

Definition 36 (Henkin Language Over A Theory)**.** The Henkin language of a theory T is the limit language constructed from the language underlying T.

Theorem 37 (Complete Henkinization)**.** *Every consistent theory admits a complete Henkin extension in its Henkin language.*

Proof. The Henkin construction yields a consistent theory with enough constants. One then applies the completion theorem from Chapter 3. Since the witness property is monotone under extension, the resulting complete theory remains Henkin. □

5.11 Present Boundary

This chapter is the current endpoint of the formalized logic-side story. The repository already reaches:

- the iterative witness-extension languages,
- the limit language L_∞,
- consistency of the finite and infinite Henkin theories,
- the existence of complete Henkin extensions of consistent theories.

What lies beyond this is not more basic Henkin machinery. The next missing pieces are the later completeness-side packaging and then the forcing and set-theoretic branch of Flypitch.

5.12 Formalization Note

The Lean file packages these constructions via declarations such as `languageStep`, `LInfty`, `witProperty`, `henkinTheoryStep`, `TInfty`, and `completeHenkinizationOfConsis`. Their mathematical role is exactly the Henkin construction described above.

Chapter 6

Status And Next Frontier

The present blueprint now tells a complete logic-side story: it starts with first-order syntax and semantics, passes through compactness and completion, develops the language-extension machinery needed for witness arguments, and ends with complete Henkin extensions of consistent theories.

6.1 What The Current Port Delivers

From a mathematical point of view, the Lean 4 repository already contains:

- a first-order proof system together with soundness,
- the theory-level notions of consistency and completeness,
- compactness in the form needed for finite-fragment arguments,
- maximal and complete consistent extensions,
- the language-colimit and reflection tools needed to manage enlarging languages,
- the Henkin construction through complete Henkin extensions.

This is already substantial. It means that the model-theoretic infrastructure behind the classical Henkin proof is no longer merely planned; it is present in verified Lean 4 code.

6.2 What Is Not Yet Present

The current state should not be overstated.

- The later completeness-side packaging beyond the present Henkin layer is not yet part of the documented Lean 4 surface.
- The forcing-side development has not yet been ported.
- The final integration leading to independence of the continuum hypothesis is therefore still absent.

So the repository has reached a meaningful stopping point on the logic side, but not the full Flypitch endpoint.

6.3 The Next Frontier

According to the current project boundary, the next major missing development is `pSet_ordinal`. This marks the beginning of the forcing-side branch.

Once that side of the project begins to land in Lean 4, the natural blueprint continuation will move from the present model-theoretic story to the set-theoretic and Boolean-valued constructions used in forcing.

6.4 How The Blueprint Should Grow

The rule for future chapters should remain the same as in this pass:

- document only mathematics that is already verified in Lean 4,
- write the chapters as mathematical explanations rather than code tours,
- use Lean names to orient the reader, not to replace the mathematics.

When the forcing branch is ported, the next chapters should therefore begin with the underlying mathematics of that branch rather than with a list of Lean modules.