

# Beyond the Plane

## Abstracting Away Geometry from AlphaGeometry

Anthony Bordg and Farzad Jafarrahmani

Huawei Lagrange Center, Paris, France  
anthony.bordg@huawei.com  
farzad.jafarrahmani@huawei.com

March 2026

**Abstract.** While AlphaGeometry has achieved gold-medalist performance on International Mathematical Olympiad geometry problems, its success remains largely confined to a specialized domain. Developing a cross-domain successor requires that we first determine the underlying logical substrate of AlphaGeometry’s domain-specific language. In this paper, we identify and propose the logic of observation as a mathematically principled language designed to serve as AlphaGeometry’s foundation. We present a formal implementation of this logic within the Lean 4 programming language. By explicitly embedding Euclidean geometry into this logic and providing a machine-checked proof of its soundness, we demonstrate how the reasoning patterns of specialized solvers can be generalized across a wide class of mathematical theories.

**Keywords:** AlphaGeometry · Logic of Observation · Lean Theorem Prover · Formal Verification · Soundness · AI Mathematician.

# Table of Contents

Beyond the Plane .....	1
<i>Anthony Bordg and Farzad Jafarrahmani</i>	
1 Introduction.....	4
1.1 AlphaGeometry .....	4
1.2 AlphaGeometry vs. AlphaProof .....	4
2 A Cross-Domain Language for AlphaGeometry .....	5
2.1 Observable Logic and Theories .....	5
Signatures. ....	5
Terms. ....	6
Observable Formulas. ....	7
Contexts.....	8
Terms-In-Context. ....	9
Formulas-In-Context. ....	9
Implementing Term and Formula Substitution. ....	10
Observable Sequents. ....	11
Observable Theories.....	12
Deduction System and Provability. ....	13
2.2 Categorical Semantics: Assigning Meaning .....	14
$\Sigma$ -Structures. ....	14
Interpretation of Terms-In-Context.....	15
Interpretation of Formulas-In-Context. ....	15
Interpretation of Substitutions.....	17
Models. ....	17
The Ordering of Subobjects.....	17
2.3 Soundness: Bridging Syntax and Semantics .....	18
Why Soundness Matters for AlphaGeometry.....	18
The Soundness Theorem. ....	18
Implementation Note. ....	19
Soundness-as-a-Service for Specialized Solvers. ....	20
2.4 Two Illustrations: Affine and Euclidean Geometries .....	20
Domain Specificity and Methodological Constraints in	
AlphaGeometry. ....	20
Affine and Euclidean Geometries.....	21
Affine Geometry. ....	21
From Geometry to Lean. ....	21
Euclidean Geometry in the Observable Fragment. ....	22
High-Level Abstractions and Diagrammatic Reasoning. ...	23
3 Conclusions and Future Work .....	24
Related Work .....	24
Formalization of First-Order Logic.....	24
Formalization of Euclidean Geometry. ....	25

	Beyond the Plane	3
	Automated Theorem Proving for Geometry. . . . .	25
	Author Contributions. . . . .	25
	Acknowledgments. . . . .	25
	Disclosure of Interests. . . . .	25
A	Deduction System. . . . .	28

## 1 Introduction

### 1.1 AlphaGeometry

AlphaGeometry is a state-of-the-art neuro-symbolic theorem prover for Euclidean plane geometry that overcomes the scarcity of human data by synthesizing millions of problems and their corresponding proofs [22]. The system’s architecture integrates a domain-specific language with a neural language model that directs a specialized symbolic deduction engine. This guidance is particularly critical for proposing auxiliary constructions (*e.g.*, new points, lines, or circles) that unlock difficult proofs.

The most recent iteration, AlphaGeometry2, achieves gold-medalist performance on International Mathematical Olympiad (IMO) geometry problems [6]. Its architecture utilizes a transformer-based model<sup>1</sup> fine-tuned on an expansive synthetic dataset. Despite its remarkable success, AlphaGeometry’s domain-specific language remains limited to a specific subset of geometry; while its coverage has improved, it currently accounts for 88% of IMO geometry problems from 2000–2024.

### 1.2 AlphaGeometry vs. AlphaProof

In the landscape of mathematical AI, AlphaProof represents a parallel engineering effort to tackle IMO-level reasoning [14]. However, the two systems operate on different philosophies. While AlphaGeometry is a neuro-symbolic specialist, AlphaProof is an AlphaZero-inspired reinforcement learning agent grounded in Lean’s formal environment [17]. The distinction between these two systems was most visible at the 2024 IMO competition. While AlphaProof served as the general-purpose reasoning engine, the Google team still relied on the specialized AlphaGeometry to solve the geometry problems [14, 6.3]. This technical partition is not rooted only in existing gaps in Mathlib’s planar geometry library, it also reflects a specialist-versus-generalist partition.

The two systems also diverge in their approach to data scarcity. On one hand, AlphaGeometry utilizes a bottom-up synthetic pipeline: it samples premises, computes a deductive closure via a symbolic engine (DD+AR), and uses a trace-back algorithm to extract minimal proof triples. On the other hand, AlphaProof utilizes a breadth-first strategy: it leverages autoformalization to convert natural language problems into Lean, and subsequently generates synthetic problem variants to create a learning curriculum that ensures local coverage around target problems.

This raises an architectural question: What would a successor to AlphaGeometry look like? We seek a system that is AlphaGeometry-inspired in its specialized efficiency, yet cross-domain and formally grounded in its execution. This article is a first attempt to define the missing quadrant in the following  $2 \times 2$  matrix.

<sup>1</sup> Either a custom transformer trained from scratch (AlphaGeometry1) or a Mixture-of-Experts model leveraging Gemini-based models pre-trained on math data and fine-tuned on an even larger set of AlphaGeometry data (AlphaGeometry2).

AlphaZero	AlphaProof
AlphaGeometry	?

Such an endeavor begins with identifying the hidden logical foundations of AlphaGeometry thereby enabling us to generalize its domain-specific language into a cross-theory substrate for the next generation of AI mathematicians.

## 2 A Cross-Domain Language for AlphaGeometry

While AlphaGeometry [22] is currently a single spoke, its direct generalization would act as the symbolic hub for a cross-domain language for AI reasoning and theorem proving. In this section, we present such a symbolic hub for AlphaGeometry via the *logic of observation* (or simply *observable logic*), a common language together with its proof engine to make AlphaGeometry reason across theories.<sup>2</sup> At the same time, we also present a machine-checked implementation of this logic in the Lean theorem prover, *i.e.* an internal representation of this object logic within Lean’s type theory.

### 2.1 Observable Logic and Theories

**Signatures.** The starting point of our implementation is the notion of a *many-sorted first-order signature*. In mathematical logic, the signature acts as the type system that defines the non-logical symbols available in our language.

A *signature*  $\Sigma$  is defined by three components:

1. **Sorts:** A set of the so-called sorts (base types), denoted  $S, T, \dots$ . These are analogous to data types or classes in programming, representing the different domains of objects we can talk about.
2. **Function symbols:** A set of symbols, each associated with an input sort list and an output sort. A function symbol  $f$  is typically written as  $f : S_1 \cdots S_n \rightarrow T$ , where  $n \in \mathbb{N}$  is its *arity* (also called *rank*). This is analogous to a function declaration, with its input type list and output type, in a programming language.
3. **Relation symbols:** A set of symbols, each associated with a finite list of sorts. A relation symbol  $R$  is typically written as  $R(S_1, \dots, S_n)$ . This is analogous to a boolean predicate or a table schema in a relational database, used to make statements that are either true or false.

The signature defines the symbols (syntax), which are then assigned meaning (semantics). In particular, a function symbol  $f : S_1 \cdots S_n \rightarrow T$  can be interpreted as an actual set-theoretic function that takes  $n$  inputs belonging to the sets

<sup>2</sup> In the mathematics literature, observable logic is often called geometric logic, a misleading terminology for a readership steeped in AlphaGeometry. Vickers in [23] uses the better, but slightly more verbose expression “the Logic of Finite Observations”.

$S_1, \dots, S_n$  respectively (i.e. an element of  $S_1 \times \dots \times S_n$ ) and returns an element of the set  $T$ , while a relation symbol  $R(S_1, \dots, S_n)$  can be interpreted as a subset of the Cartesian product  $S_1 \times \dots \times S_n$  (the relation  $R$  holds true for a tuple of elements if and only if that tuple is a member of this subset).

We implement this definition as follows in Lean.

```
structure Signature where
  Base : Type w'
  Index : Type w''
  leftIndex : Index
  rightIndex : Index
  Sorts : Type u
  newLabel : (X : Finset (Base × Sorts)) → {x : Base | ∀ v, v ∈ X → x ≠ v.1}
  Fun : Type v
  Rel : Type w
  rankF : Fun → List Sorts × Sorts
  rankR : Rel → List Sorts
```

The internal representation of signatures within Lean’s type theory, using a `structure` (a record in Lean), presents more fields than one may expect from the above definition, but the reader will recognize that the core constructors `Sorts`, `Fun`, `Rel`, and `rankF` / `rankR` correspond to sorts, function symbols, relation symbols and their arities, respectively. We will model a variable as a pair made of a label and its sort, and since we want an infinite, possibly uncountable, set of variables for every sort  $S$ , we introduce a base type `Base` for indexing labels (instead of using a ready-made type, e.g.  $\mathbb{N}$ ). The constructor `newLabel` simply makes sure that in any context, i.e. for any finite set of variables, we are free to introduce a new variable, namely a new label of some sort  $S$ . The additional fields in the Lean snippet will become clear when we introduce formulas.

**Terms.** We now define the collection of sorted *terms*. Terms will become the basic building blocks of our logical statements. We formally define the collection of all valid sorted terms over a signature  $\Sigma$  using the following inductive rules:

1. **Variables (base case):** Every variable  $x$ , which is declared to be of some sort  $S$  (denoted  $x : S$ ), is a term of sort  $S$ .
2. **Function Application (inductive step):** For every list of existing terms  $t_1, \dots, t_n$ , where each term  $t_i$  has the corresponding sort  $S_i$  (i.e.,  $t_i : S_i$ ), and every function symbol  $f : S_1 \dots S_n \rightarrow T$  from  $\Sigma$ , the function application  $f(t_1, \dots, t_n)$  is a new term of sort  $T$ .

In Lean, sorted terms are coded as an inductive family of types parametrized by the type `Sorts` of sorts. This ensures that every term is associated with a unique sort from our signature.

```

inductive Term :  $\sigma$ .Sorts  $\rightarrow$  Type max u v w'
| var (x :  $\sigma$ .Base  $\times$   $\sigma$ .Sorts) : Term x.2
| app f (arg : (i : Fin ( $\sigma$ .rankF f).1.length)  $\rightarrow$  Term (( $\sigma$ .rankF f).1.get i)) :
  Term ( $\sigma$ .rankF f).2

```

Throughout this paper, all terms are understood to be sorted terms within our many-sorted framework.

**Observable Formulas.** Next, we introduce the concept of logical *formulas* (often called *expressions* or *propositions* in programming) built from the terms we just defined. These formulas are the statements that can be evaluated to a truth value. We restrict our attention to a well-known, highly structured subclass  $\mathcal{F}$  of first-order formulas that we call *observable* formulas. Observable formulas are defined inductively from *atomic* formulas using the following three logical connectors: binary (hence, finitary) conjunctions ( $\wedge$ ), infinitary disjunctions ( $\bigvee$ ), and existential quantifications ( $\exists$ ).

Atomic formulas are the smallest, non-decomposable expressions. They are defined directly from the terms and relation symbols of the signature  $\Sigma$ . Crucially, for every formula  $\phi$ , we track its set of free variables,  $\text{FV}(\phi)$ , which are the variables not bound by a quantifier.

1. **Relations (predicates):**  $R(t_1, \dots, t_n)$  is in  $\mathcal{F}$  if  $R(S_1, \dots, S_n)$  is a relation symbol and  $t_1 : S_1, \dots, t_n : S_n$  are terms; and  $\text{FV}(R(t_1, \dots, t_n))$  is the union of all variables occurring in the terms  $t_i$ .
2. **Equality:**  $t = u$  is in  $\mathcal{F}$  if  $t$  and  $u$  are two terms of the same sort;  $\text{FV}(t = u)$  is the union of all variables occurring in  $t$  or  $u$ .

We also add the unconditional True and False propositions.

3. **Truth:**  $\top$  is in  $\mathcal{F}$ ;  $\text{FV}(\top) = \emptyset$ .
4. **Falsum:**  $\perp$  is in  $\mathcal{F}$ ;  $\text{FV}(\perp) = \emptyset$ .

The class  $\mathcal{F}$  of all observable formulas is the smallest set that contains all atomic formulas,  $\top$ ,  $\perp$ , and is closed under the following inductive composition rules (the rules also define how the set of free variables is inherited):

5. **Binary conjunction:**  $\phi \wedge \psi$  is in  $\mathcal{F}$  if  $\phi$  and  $\psi$  are in  $\mathcal{F}$ ;  $\text{FV}(\phi \wedge \psi) = \text{FV}(\phi) \cup \text{FV}(\psi)$ .
6. **Infinitary disjunction:**  $\bigvee_{i \in I} \phi_i$  is in  $\mathcal{F}$  if  $I$  is a set,  $\phi_i$  is in  $\mathcal{F}$  for each  $i \in I$ ; and  $\text{FV}(\bigvee_{i \in I} \phi_i) = \bigcup_{i \in I} \text{FV}(\phi_i)$ .
7. **Existential quantification:**  $(\exists x)\phi$  is in  $\mathcal{F}$  if  $\phi$  is in  $\mathcal{F}$  and  $x$  is a variable;  $\text{FV}((\exists x)\phi) = \text{FV}(\phi) \setminus x$ .

This inductive definition provides the complete syntax for the class of observable formulas we will be working with. Since this definition is complicated by the fact we are simultaneously defining formulas and their free variables, in Lean we will first define formulas and handle separately their free variables through the introduction of a gadget called a *context*.

```

inductive Formula : Type _
| truth : Formula
| falsum : Formula
| equal s :  $\sigma$ .Term s  $\rightarrow$   $\sigma$ .Term s  $\rightarrow$  Formula
| binConj : Formula  $\rightarrow$  Formula  $\rightarrow$  Formula
| infDisj : (I : Set  $\sigma$ .Index)  $\rightarrow$  (I  $\rightarrow$  Formula)  $\rightarrow$  Formula
| exist :  $\sigma$ .Base  $\times$   $\sigma$ .Sorts  $\rightarrow$  Formula  $\rightarrow$  Formula
| rel R (arg : (i : Fin ( $\sigma$ .rankR R).length)  $\rightarrow$   $\sigma$ .Term (( $\sigma$ .rankR R).get i)) :
    Formula

notation:100 "T" => Formula.truth
notation:100 "F" => Formula.falsum
notation:110 t " = " u:110 " : " s:110 => Formula.equal s t u
infix:120 " ^ " => Formula.binConj
notation:120 "V " I " , " fs => Formula.infDisj I fs
notation:110 "( $\exists$  " x:110 " ) "  $\phi$ :110 => Formula.exist x  $\phi$ 

```

**Contexts.** In our formalization, we must ensure that the set of free variables,  $FV(\phi)$ , is always finite for any given formula  $\phi$ , and that we can strictly control and track these variables. This requirement should also hold for infinite disjunctions  $\bigvee_{i \in I} \phi_i$ , where  $I$  can be of an arbitrary cardinality. To achieve this control, we introduce the notion of a *context* and the associated concepts of *term-in-context* and *formula-in-context*. By forcing every formula to be defined within a specific context, we guarantee that all its free variables are explicitly declared and form a finite set. This step allows us to transition from the raw syntax of formulas to the well-defined and verifiable syntax required for the Lean implementation.

In Lean, we model a context as a data structure that captures the necessary information: the labels and the sorts of the variables. More precisely, a context is represented as a finite set of sorted variables, where a sorted variable is a pair consisting of a label (the variable name, e.g.,  $x$ ) and its sort (e.g.,  $S$ ). We use Lean’s `Finset` data type for this, which guarantees the following key properties:

- Finiteness: a context is finite.
- No order and no duplicate elements: in Lean, finite sets are multisets, i.e. lists up to permutation, which have no duplicate elements.

However, since a sorted label is a pair  $(x, S)$ , two distinct entries can still have the same label but different sorts, such as  $(x, \text{Integer})$  and  $(x, \text{String})$ . To enforce a strict control over variable labels, we require that every label within a single context must refer to only one sort. To this effect, we define a dedicated structure `Ctx` for our context that wraps Lean’s `Finset` with an additional requirement (an additional field) to ensure label uniqueness.

```

structure Ctx where
  ctx : Finset ( $\sigma$ .Base  $\times$   $\sigma$ .Sorts)
  single_sort :  $\forall$  x y, x  $\in$  ctx  $\rightarrow$  y  $\in$  ctx  $\rightarrow$  x.1 = y.1  $\rightarrow$  x.2 = y.2

```

**Terms-In-Context.** We connect terms and contexts through the notion of a *term-in-context*. This judgment, written formally as  $\Gamma \vdash t$ , signifies that the context  $\Gamma$  contains declarations for all variables appearing in  $t$ . In our Lean implementation, this is captured by a proposition<sup>3</sup> where the sort  $s$  of the term  $t$  is treated as an implicit argument (indicated by curly braces  $\{s : \text{Sort}\}$ ). This allows Lean’s elaborator to infer the sort automatically from the type of  $t$ , maintaining a concise syntax that mirrors informal mathematical practice. The predicate is defined by induction on the structure of  $t$  in a straightforward manner.

```
def Term.InCtx (Γ : σ.Ctx) {s : σ.Sorts} (t : σ.Term s) : Prop :=
  match t with
  | var x => x ∈ Γ.ctx
  | app f ts => ∀ (i : Fin (σ.rankF f).1.length), (ts i).InCtx Γ

notation:80 Γ " ⊢ " t:90 => Term.InCtx Γ t
```

**Formulas-In-Context.** We connect formulas and contexts using the notion of a *formula-in-context*, written formally as  $\Gamma \vdash \phi$ . This notation signifies that the formula  $\phi$  should be understood in relation with a context  $\Gamma$ . Specifically, the judgment  $\Gamma \vdash \phi$  implies two conditions are met:

1. **Free Variables are Declared:** The context  $\Gamma$  contains declarations for all of the free variables  $\text{FV}(\phi)$  of the formula  $\phi$ .
2. **No Bound Variable Conflicts:** The context  $\Gamma$  does not contain any of the variables that are bound by a quantifier within  $\phi$ .

In our Lean implementation, this judgment is captured by the predicate `InCtx`, which enforces these typing and scoping constraints. The relationship between quantification and context is inductive: for a formula  $(\exists x)\phi$  to be well-formed in context  $\Gamma$ , the sub-formula  $\phi$  must be well-formed in context  $\Gamma$  extended by the variable  $x$ . In our formalization, a context  $\Gamma$  is extended by a variable  $x$  (of a given sort) by providing a proof  $p$  that the label  $x$  is fresh, i.e., it does not already appear in  $\Gamma$ . This extended context is denoted  $\Gamma :: \langle x, p \rangle$ . Consequently, the formation rule for existence requires that  $(\exists x)\phi$  is in context  $\Gamma$  if there exists a proof  $p$  (i.e., a witness) such that  $\phi$  is in context  $\Gamma :: \langle x, p \rangle$ . This ensures by construction that no bound variable  $x$  can conflict with the existing declarations in  $\Gamma$ .

```
def extended (y : σ.Base × σ.Sorts) (p : y.1 ∉ Γ.Label) : σ.Ctx where
  ctx := insert y Γ.ctx
  single_sort x x' p q eq := by ...
```

```
notation:110 Γ " :: " " ⟨ y:110 ", " p:110 " ⟩" => extended Γ y p
```

```
def Formula.InCtx (Γ : σ.Ctx) (φ : σ.Formula) : Prop :=
```

<sup>3</sup> Technically, a term of type `Prop`, Lean’s universe of propositions.

```

match  $\varphi$  with
|  $\top$  => True
|  $\perp$  => True
|  $t = u : \_$  =>  $\Gamma \vdash t \wedge \Gamma \vdash u$ 
|  $\varphi \wedge' \psi$  =>  $\varphi.\text{InCtx } \Gamma \wedge \psi.\text{InCtx } \Gamma$ 
|  $\vee I, fs$  =>  $(\forall i : I, (fs\ i).\text{InCtx } \Gamma) \wedge (\vee I, fs).\text{freeVar} \subseteq \Gamma.\text{ctx}$ 
|  $(\exists' x)\varphi$  =>  $\exists p : x.1 \notin \Gamma.\text{Label}, \varphi.\text{InCtx } (\Gamma :: (x, p))$ 
|  $.\text{rel } \_ ts$  =>  $\forall i, \Gamma \vdash ts\ i$ 

infix:80 "  $\vdash$  " => Formula.InCtx

```

We then check in Lean that the two aforementioned conditions are satisfied.

```

theorem freeVar_subset_ctx_of_InCtx :
   $\forall \{\varphi : \sigma.\text{Formula}\} \{\Gamma : \sigma.\text{Ctx}\}, \Gamma \vdash \varphi \rightarrow \varphi.\text{freeVar} \subseteq \Gamma.\text{ctx}$ 

theorem boundVar_inter_ctx_empty :
   $\forall \{\varphi : \sigma.\text{Formula}\} \{\Gamma : \sigma.\text{Ctx}\}, \Gamma \vdash \varphi \rightarrow \varphi.\text{boundVar} \cap \Gamma.\text{ctx} = \emptyset$ 

```

As a final remark for this section, the intuition behind the logic of observation is most transparent in its inherent asymmetry between finite conjunctions and infinite disjunctions. In any given context, a property is confirmed through a finite sequence of “tests” (conjunctions), even if that property can be satisfied in an infinite number of potential ways (disjunctions) [23]. It is precisely this asymmetry that characterizes the logic of observation and one aspect of its suitability for machine-checked reasoning.

**Implementing Term and Formula Substitution.** In our Lean-based implementation, we must rigorously define the operations for substituting terms for variables. This involves two distinct layers: first, the substitution of terms for variables within a given term to generate a more complex term; and second, the substitution of terms for the free variables within a formula to specialize a logical assertion.

To formalize these operations, we define the notion of a *substitution between two contexts*.<sup>4</sup> A substitution  $\gamma$  between two contexts  $\Gamma$  and  $\Delta$ , denoted  $\gamma : \text{Subst}(\Gamma, \Delta)$ , is a mapping that assigns to each variable  $x_i$  in the source context  $\Gamma$  a term  $t_i$  of the same sort, constructed using the variables available in the target context  $\Delta$  (i.e. in such a way that every  $t_i$  is a sorted term in context  $\Delta$ , formally  $\Delta \vdash t_i$ ). In Lean, a substitution is implemented as a structure parametrized by these two contexts. This structure has two fields enforcing the sorted nature of the signature and the above requirements, ensuring it is a “type-safe” structure.

```

structure Subst ( $\Gamma \Delta : \sigma.\text{Ctx}$ ) where
  map : ( $x : \Gamma.\text{ctx}$ )  $\rightarrow \sigma.\text{Term } x.1.2$ 
  inCtx_map  $x : \Delta \vdash \text{map } x$ 

```

<sup>4</sup> For a comprehensive background on the formal treatment of abstract syntax and variable binding, see [10].

Given a term  $t$  in context  $\Gamma$ , the specialization of that term according to a substitution  $\gamma : \text{Subst}(\Gamma, \Delta)$  is defined through pattern-matching on the structure of  $t$ . This process essentially replaces every variable occurrence within the term with its corresponding image under  $\gamma$ .

```
def term {s :  $\sigma$ .Sorts} (t :  $\sigma$ .Term s) (p :  $\Gamma \vdash t$ ) :  $\sigma$ .Term s :=
  match t with
  | .var x =>  $\gamma$ .map (x, p)
  | .app f ts => f (fun i => term (ts i) (p i))
```

We formally prove that the resulting term, denoted  $t[\gamma]$ , is a valid term within the target context  $\Delta$  (formally,  $\Delta \vdash t[\gamma]$ ).

```
theorem term_inCtx {s :  $\sigma$ .Sorts} {t :  $\sigma$ .Term s} (h :  $\Gamma \vdash t$ ) :  $\Delta \vdash t[\gamma]$  h
```

Finally, given a formula  $\phi$  in context  $\Gamma$ , the specialization of that formula according to a substitution  $\gamma : \text{Subst}(\Gamma, \Delta)$  is defined through structural pattern-matching. For atomic formulas, such as the equality of terms ( $t = u$ ), the process invokes the previously defined substitution operation on terms. The complexity arises in the case of quantified formulas, specifically when  $\phi := (\exists x)\psi$ . To maintain logical soundness, the system must prevent the capture of free variables. If the variable  $x$  (the binder) already exists in the target context  $\Delta$  and, as a result, possibly appears within the terms of the substitution  $\gamma$ , it is renamed within  $\Delta$  and the affected terms before the substitution is applied to the sub-formula  $\psi$ .

```
def formula { $\Gamma \Delta$  :  $\sigma$ .Ctx} ( $\gamma$  :  $\Gamma$ .Subst  $\Delta$ ) ( $\phi$  :  $\sigma$ .Formula) (p :  $\Gamma \vdash \phi$ ) :
   $\sigma$ .Formula :=
  match  $\phi$  with
  |  $\top$  =>  $\top$ 
  |  $\perp$  =>  $\perp$ 
  |  $t = u$  : s => (t[ $\gamma$ ] p.1) = (u[ $\gamma$ ] p.2) : s
  |  $\phi \wedge \psi$  => ( $\gamma$ .formula  $\phi$  p.1)  $\wedge$  ( $\gamma$ .formula  $\psi$  p.2)
  |  $\forall I$ , fs =>  $\forall I$ , fun i =>  $\gamma$ .formula (fs i) (p.1 i)
  | ( $\exists x$ ) $\phi$  => if h : x.1  $\in$   $\Delta$ .Label
    then ( $\exists$ '( $\sigma$ .newLabel  $\Delta$ .ctx, x.2))
      ( $\gamma$ .extended p.1  $\Delta$ .newLabel_notin_label x.2).formula  $\phi$  p.2
    else ( $\exists x$ ) ( $\gamma$ .extended p.1 h x.2).formula  $\phi$  p.2
  | rel R ts => rel R (fun i => (ts i)[ $\gamma$ ] (p i))
```

```
notation:101  $\phi$  " ["  $\gamma$  "]" " p => formula  $\gamma$   $\phi$  p
```

We formally prove that the resulting formula, denoted  $\phi[\gamma]$ , is a valid formula within the target context  $\Delta$  (formally,  $\Delta \vdash \phi[\gamma]$ ).

```
theorem formula_inCtx :
   $\forall$  { $\phi$  :  $\sigma$ .Formula} { $\Gamma \Delta$  :  $\sigma$ .Ctx} ( $\gamma$  :  $\Gamma$ .Subst  $\Delta$ ) (p :  $\Gamma \vdash \phi$ ),  $\Delta \vdash \phi[\gamma]$  p
```

**Observable Sequents.** A key limitation of observable formulas is the lack of universal quantification ( $\forall$ ) and standard implication ( $\implies$ ). Thus, we cannot directly express a common logical statement like:

$$(\forall x_1) \dots (\forall x_n) (\phi \implies \psi)$$

This formula means: “For all values assigned to the variables  $x_i$ , if the premise  $\phi$  is true, then the conclusion  $\psi$  must also be true.” To express this necessary relationship within our restricted logic, we introduce a formal expression called an *observable sequent* over the signature  $\Sigma$ . A sequent is written in the form:

$$\phi \vdash_{\vec{x}} \psi,$$

where  $\phi$  (the premise) and  $\psi$  (the conclusion) are observable formulas over  $\Sigma$ , and  $\vec{x}$  is a context that contains all of the free variables in both  $\phi$  and  $\psi$  (assumed to be the  $x_i$  here, hence the notation for the context). This formal expression implicitly introduces the necessary universal quantification over these free variables. The sequent  $\phi \vdash_{\vec{x}} \psi$  is the exact formal expression of the excluded universally quantified implication. Its intended logical meaning is that  $\psi$  is a logical consequence of  $\phi$  in the context  $\vec{x}$ .

```
structure Sequent where
  ctx :  $\sigma$ .Ctx
  left :  $\sigma$ .Formula
  right :  $\sigma$ .Formula
  isFormula_left : ctx  $\vdash$  left
  isFormula_right : ctx  $\vdash$  right
```

It is important to note that this structure is “flat”: while the sequent itself acts as a top-level universally quantified implication, the formulas  $\phi$  and  $\psi$  remain strictly within the observable fragment, meaning they cannot themselves contain further implications or universal quantifiers.

**Observable Theories.** We now arrive at the central concept: *observable theories* over a signature  $\Sigma$ . An observable theory is a complete specification or axiomatization of a mathematical structure within the language defined by  $\Sigma$ . By an observable theory  $\mathbb{T}$  over a signature  $\Sigma$ , we mean simply a set of observable sequents over  $\Sigma$ .

```
structure Theory where
  axioms : Set  $\sigma$ .Sequent
```

The sequents in  $\mathbb{T}$  act as the axioms, namely the non-negotiable rules or conditions that the intended mathematical structure must satisfy. Importantly, observable theories are highly expressive and encompass the axiomatizations of a vast array of mathematical concepts. Examples of structures that can be fully captured by observable theories include among others:

- **Classical Geometries:** Euclidean geometry, affine geometry.
- **Algebraic Structures:** monoids, groups, abelian groups, rings, modules over a ring, and fields.
- **Abstract Structures:** the theory of small categories and directed graphs.

**Deduction System and Provability.** The logic of observation is not only a convenient way to define mathematical structures (theories  $\mathbb{T}$ ), but it also provides a complete *deduction system*: a formal mechanism to prove theorems about them. This system is analogous to a set of inference rules in an automated theorem prover or a rewriting system. This deduction system is composed of specific rules for:

- **Structural Rules:** For manipulating the layout of sequents (e.g., weakening or cutting).
- **Logical Connectives:** For introducing and eliminating conjunctions, infinitary disjunctions, and the existential quantifier.
- **Equality:** For handling the properties of the equality predicate.

The system allows us, in some context  $\Gamma$ , to deduce a new assertion ( $\psi$ ) from an existing one ( $\phi$ ) through a chain of intermediate steps, by repeatedly applying these rules. Such a chain of rule applications is formally known as a *proof tree* (or a *proof* for short). Crucially, a proof is always defined relative to an observable theory  $\mathbb{T}$ . This means the proof is allowed to use any of the sequents (axioms) defined in  $\mathbb{T}$  as starting points or steps, alongside the system’s core logical rules. For a complete reference, the core logical rules of this deduction system are provided in Appendix A. In the Lean implementation that follows, the type of proof trees is modeled as an inductive data type, mirroring the recursive structure of the proof process.

```

inductive Proof :  $\sigma$ .Ctx  $\rightarrow$   $\sigma$ .Formula  $\rightarrow$   $\sigma$ .Formula  $\rightarrow$  Type _
| ax : (seq :  $\sigma$ .Sequent)  $\rightarrow$  seq  $\in$  T.axioms  $\rightarrow$  Proof seq.ctx seq.left seq.right
| id  $\Gamma$  ( $\phi$  :  $\sigma$ .Formula) : Proof  $\Gamma$   $\phi$   $\phi$ 
| subst { $\Gamma$   $\Delta$  :  $\sigma$ .Ctx} ( $\gamma$  :  $\Gamma$ .Subst  $\Delta$ ) :
  Proof  $\Gamma$   $\phi$   $\psi \rightarrow$  ( $p$  :  $\Gamma \vdash \phi$ )  $\rightarrow$  ( $q$  :  $\Gamma \vdash \psi$ )  $\rightarrow$  Proof  $\Delta$  ( $\phi[\gamma]$   $p$ ) ( $\psi[\gamma]$   $q$ )
| cut  $\Gamma$   $\phi$  ( $\psi$  :  $\sigma$ .Formula)  $\chi$  ( $h$  :  $\Gamma \vdash \psi$ ) : Proof  $\Gamma$   $\phi$   $\psi \rightarrow$  Proof  $\Gamma$   $\psi$   $\chi \rightarrow$  Proof  $\Gamma$   $\phi$   $\chi$ 
| top  $\Gamma$   $\phi$  : Proof  $\Gamma$   $\phi$  ( $\top$ )
| conjElimLeft  $\Gamma$   $\phi$   $\psi$  : Proof  $\Gamma$  ( $\phi \wedge' \psi$ )  $\phi$ 
| conjElimRight  $\Gamma$   $\phi$   $\psi$  : Proof  $\Gamma$  ( $\phi \wedge' \psi$ )  $\psi$ 
| conjIntro  $\Gamma$   $\phi$   $\psi$   $\chi$  : Proof  $\Gamma$   $\phi$   $\psi \rightarrow$  Proof  $\Gamma$   $\phi$   $\chi \rightarrow$  Proof  $\Gamma$   $\phi$  ( $\psi \wedge' \chi$ )
| bot  $\Gamma$   $\phi$  : Proof  $\Gamma$  ( $\perp$ )  $\phi$ 
| infDisjIntro  $\Gamma$  I fs i : Proof  $\Gamma$  (fs i) ( $\vee$  I, fs)
| disL  $\Gamma$  I fs  $\phi$  : ( $\forall$  i, Proof  $\Gamma$  (fs i)  $\phi$ )  $\rightarrow$  Proof  $\Gamma$  ( $\vee$  I, fs)  $\phi$ 
| existIntro  $\Gamma$  y p  $\phi$  ( $\psi$  :  $\sigma$ .Formula) ( $h$  : ( $\Gamma :: \langle y, p \rangle$ )  $\vdash \psi$ ) :
  Proof ( $\Gamma :: \langle y, p \rangle$ )  $\phi$   $\psi \rightarrow$  Proof  $\Gamma$  ( $\exists'y$ )  $\phi$   $\psi$ 
| existElim  $\Gamma$  y p  $\phi$  ( $\psi$  :  $\sigma$ .Formula) ( $h$  :  $\Gamma \vdash \psi$ ) :
  Proof  $\Gamma$  ( $\exists'y$ )  $\phi$   $\psi \rightarrow$  Proof ( $\Gamma :: \langle y, p \rangle$ )  $\phi$   $\psi$ 
| distributive  $\Gamma$   $\phi$  I fs : Proof  $\Gamma$  ( $\phi \wedge' (\vee$  I, fs)) ( $\vee$  I, fun i  $\Rightarrow$   $\phi \wedge' fs$  i)
| frobenius  $\Gamma$   $\phi$  x  $\psi$  : Proof  $\Gamma$  ( $\phi \wedge' ((\exists'x) \psi)$ ) ( $((\exists'x) \phi \wedge' \psi)$ )
| substOfEq ( $\Gamma_1$   $\Gamma_2$   $\Delta$  :  $\sigma$ .Ctx) ( $p$  :  $\Gamma_1$ .Renaming  $\Gamma_2$ ) ( $\phi$  :  $\sigma$ .Formula) ( $p$  :  $\Gamma_1 \vdash \phi$ )
  ( $h1$  :  $\Gamma_1.ctx \subseteq \Delta.ctx$ ) ( $h2$  :  $\Gamma_2.ctx \subseteq \Delta.ctx$ ) :
  Proof  $\Delta$  (finConj p.listEq  $\wedge' \phi$ ) ( $\phi[p.substOfRenaming]$   $p$ )

```

Finally, we use the deduction system to define the core concept of *provability*. A sequent  $\sigma$  is said to be *provable* in a theory  $\mathbb{T}$  or  *$\mathbb{T}$ -provable* (written  $\mathbb{T} \vdash \sigma$ )

if and only if there exists a valid proof tree relative to the axioms of  $\mathbb{T}$  that concludes with  $\sigma$ . In the context of the Lean type system, this is equivalent to stating that the corresponding type `T.Proof  $\sigma$`  of proof trees is non-empty: meaning we have successfully constructed a proof term belonging to that type.

```
def Provable (seq :  $\sigma$ .Sequent) : Prop :=
  Nonempty (T.Proof seq.ctx seq.left seq.right)
```

This completes the formal definition of observable theories and their associated deduction system.

## 2.2 Categorical Semantics: Assigning Meaning

The deduction rules presented in Appendix A provide a complete syntactic foundation for the logic of observation, defining the formal notion of provability in our system. While these rules are purely formal, they were deliberately conceived to align with a robust *semantic* (meaning-based) foundation. This semantic foundation requires a notion of **model**: a class of mathematical structures where the expressions of our language can be interpreted, giving rise to the concept of **satisfaction**. Specifically, our approach assigns meaning to the expressions of our language in a **category** (in the sense of category theory). This is a direct generalization of Tarski’s satisfaction of first-order formulas, which traditionally interprets expressions in set-valued structures (i.e., assigning truth values based on sets). As a result, this section will be devoted to defining these crucial notions of model and of satisfaction of a sequent in such a model.

**$\Sigma$ -Structures.** To interpret a signature  $\Sigma$  in a category  $\mathcal{C}$ , we define the notion of a  $\Sigma$ -structure  $M$ . Think of  $M$  as a realization or deployment of the abstract signature into a specific mathematical environment, much like the implementation of an interface in programming.

Formally, a  $\Sigma$ -structure  $M$  in a category  $\mathcal{C}$  (with finite products) is a mapping that assigns:

1. To each sort  $S$ : An object  $MS$  in  $\mathcal{C}$ . (The “data type” becomes a “space” or a generalized set). To accommodate our many-sorted setting, this mapping is extended to finite sequences of sorts  $S_1 \cdots S_n$  by defining  $M(S_1, \dots, S_n) := MS_1 \times \cdots \times MS_n$ .
2. To each function symbol  $(f : S_1 \dots S_n \rightarrow T)$ : A morphism (arrow)

$$Mf : MS_1 \times \cdots \times MS_n \rightarrow MT$$

in  $\mathcal{C}$ .

3. To each relation symbol  $R(S_1, \dots, S_n)$ : A subobject of the product  $MS_1 \times \cdots \times MS_n$ , written  $MR \multimap MS_1 \times \cdots \times MS_n$ . (A subobject is a categorical generalization of a subset, see Section 2.2 later for more details).

In Lean, we implement this as a structure that packages in a concise way these mappings together, allowing us to treat a  $\Sigma$ -structure as a first-class object in our formalization.

```

structure Str where
  mapSort :  $\sigma$ .Sorts  $\rightarrow$  C
  mapFun f :  $\prod^c$  (fun n => mapSort (( $\sigma$ .rankF f).1.get n))  $\rightarrow$  mapSort ( $\sigma$ .rankF f).2
  mapRel R : MonoOver ( $\prod^c$  (fun n => mapSort (( $\sigma$ .rankR R).get n)))

notation:110 M:110s " s:110 => Str.mapSort M s
notation:110 M:110f " f:110 => Str.mapFun M f
notation:110 M:110r " R:110 => Str.mapRel M R

```

**Interpretation of Terms-In-Context.** Let  $M$  be a  $\Sigma$ -structure over  $\mathcal{C}$  and  $\Gamma \vdash t$  a term-in-context over  $\Sigma$ , with  $\Gamma := x_1 : S_1, \dots, x_n : S_n$  and  $t : T$ . The interpretation of the context itself, denoted  $M\Gamma$ , is the product:

$$M\Gamma := MS_1 \times \dots \times MS_n.$$

```

def Ctx.intrpIn ( $\Gamma$  :  $\sigma$ .Ctx) : C :=  $\prod^c$  (fun x :  $\Gamma$ .ctx  $\mapsto$  Ms x.1.2)

```

We define the interpretation  $Mt^\Gamma$  of  $t$  in context  $\Gamma$  as a morphism

$$M\Gamma \rightarrow MT$$

by induction, as follows.

1. If  $t$  is one of the  $x_i$ , then  $Mt^\Gamma$  is the  $i$ th projection.
2. If  $t := f(t_1, \dots, t_m)$  (where  $t_i : U_i$  in context  $\Gamma$ , say), then  $Mt^\Gamma$  is the composite

$$M\Gamma \xrightarrow{(Mt_1^\Gamma, \dots, Mt_m^\Gamma)} M(U_1, \dots, U_m) \xrightarrow{Mf} MT.$$

```

def Term.intrpIn {s :  $\sigma$ .Sorts} (t :  $\sigma$ .Term s) (p :  $\Gamma \vdash t$ ) :  $\Gamma$ .intrpIn M  $\rightarrow$  Ms s :=
  match t with
  | var x =>  $\Gamma$ .intrpInn M (x,p)
  | app f ts => Pi.lift (fun i  $\mapsto$  (ts i).intrpIn (p i))  $\gg$  Mf f

```

**Interpretation of Formulas-In-Context.** Given a  $\Sigma$ -structure  $M$  over  $\mathcal{C}$ , we can interpret all formulas-in-context in the category  $\mathcal{C}$  provided  $\mathcal{C}$  has enough structure.<sup>5</sup> Let  $\phi^\Gamma$  be a formula in the context  $\Gamma := x_1 : S_1, \dots, x_n : S_n$ . We define the interpretation  $M\phi^\Gamma$  of  $\phi^\Gamma$  as a subobject of  $M\Gamma := MS_1 \times \dots \times MS_n$ .<sup>6</sup> For this interpretation to be possible,  $\mathcal{C}$  must possess “enough structure” to handle the logical operations defined in our formulas. Specifically,  $\mathcal{C}$  must support the categorical equivalents of finite conjunctions, infinite disjunctions, and existential quantification.

We proceed by induction on the structure of  $\phi^\Gamma$ , starting with the atomic cases.

<sup>5</sup> Technically, if  $\mathcal{C}$  is an observable category (often called a geometric category in the mathematics literature, see [15, A 1.4.18]).

<sup>6</sup> While this subobject is strictly an isomorphism class of monomorphisms  $M\phi \hookrightarrow M\Gamma$ , we follow the standard abuse of notation by referring to the subobject simply by a representative domain  $M\phi$ ; and in the formalization we directly work with a representative monomorphism using Mathlib’s `MonoOver`, the category of monomorphisms into an object.

1. If  $\phi^\Gamma := \top$ , then  $M\phi^\Gamma$  is the largest subobject of  $M\Gamma$ .
2. If  $\phi^\Gamma := \perp$ , then  $M\phi^\Gamma$  is the smallest subobject of  $M\Gamma$ .
3. If  $\phi^\Gamma := R(t_1, \dots, t_m)$  where  $R := R(T_1, \dots, T_m)$  is a relation symbol, then the subobject  $M\phi^\Gamma$  is deduced from the subobject  $MR$  by base change along the morphism  $(Mt_1^\Gamma, \dots, Mt_m^\Gamma)$ .

$$\begin{array}{ccc} M\phi^\Gamma & \xrightarrow{\quad} & MR \\ \downarrow & & \downarrow \\ M\Gamma & \xrightarrow{(Mt_1^\Gamma, \dots, Mt_m^\Gamma)} & M(T_1, \dots, T_m) \end{array}$$

4. If  $\phi^\Gamma := (t = u)$  is an equality between two terms  $t$  and  $u$  of some sort  $T$  in context  $\Gamma$ , then  $M\phi^\Gamma$  is the equalizer of the following diagram.

$$M\Gamma \begin{array}{c} \xrightarrow{Mt^\Gamma} \\ \xleftarrow{Mu^\Gamma} \end{array} MT$$

5. If  $\phi^\Gamma := \psi \wedge \chi$ , then  $M\phi^\Gamma$  is the intersection (i.e. fiber product) of the subobjects  $M\psi^\Gamma$  and  $M\chi^\Gamma$ .

$$\begin{array}{ccc} M\phi^\Gamma & \rightharpoonup & M\psi^\Gamma \\ \downarrow & & \downarrow \\ M\chi^\Gamma & \rightharpoonup & M\Gamma \end{array}$$

6. If  $\phi^\Gamma := \bigvee_{i \in I} \phi_i$ , then  $M\phi^\Gamma$  is the union of the subobjects  $M\phi_i^\Gamma$ .
7. If  $\phi^\Gamma := (\exists y)\psi$  where  $y$  is of sort  $T$ , then  $M\phi^\Gamma$  is the image of the composite

$$M\psi^{\Gamma, T} \rightharpoonup M(S_1, \dots, S_n, T) \xrightarrow{\pi} M(S_1, \dots, S_n)$$

where  $\pi$  is the projection on the first  $n$  factors.

```
def Formula.intrpIn {Γ : σ.Ctx} (φ : σ.Formula) (p : Γ ⊢ φ) :
  MonoOver (Γ.intrpIn M) :=
  match φ with
  | ⊤ => ⊤
  | ⊥ => ⊥_ (MonoOver (Γ.intrpIn M))
  | t1 = t2 : _ => MonoOver.mk' (equalizer.1 (t1.intrpIn M p.1) (t2.intrpIn M p.2))
  | φ1 ∧ φ2 => (MonoOver.inf.obj (φ1.intrpIn p.1)).obj (φ2.intrpIn p.2)
  | ∀ I, fs => sigmaObj (fun i : I => (fs i).intrpIn (p.1 i))
  | (∃ x) φ => (MonoOver.«exists» ((Γ.substn x p.1).intrpIn M)).obj (φ.intrpIn p.2)
  | .rel R ts =>
    (MonoOver.pullback (Pi.lift (fun i => (ts i).intrpIn M (p i))))).obj (M' R)
```

**Interpretation of Substitutions.** Given a  $\Sigma$ -structure  $M$  over  $\mathcal{C}$  and a substitution  $\gamma : \text{Subst}(\Gamma, \Delta)$  between contexts  $\Gamma$  and  $\Delta$ , the interpretation  $M\gamma$  is the unique morphism  $M\Delta \rightarrow M\Gamma$  induced by the universal property of the product. Specifically, it is the canonical map into the product  $M\Gamma$  determined by the interpretations of the individual terms comprising  $\gamma$ .

```
def Subst.intrpIn {Δ : σ.Ctx} (γ : Γ.Subst Δ) : Δ.intrpIn M → Γ.intrpIn M :=
  Pi.lift (fun x : Γ.ctx → (γ.map x).intrpIn M (γ.inCtx_map x))
```

**Models.** To compare the truth of different formulas, we need a way to determine if one interpretation is “contained” within another. In set theory, we use subset inclusion ( $\subseteq$ ) when stating that “the set of assignments satisfying a formula  $\phi$  is a subset of the assignments satisfying a formula  $\psi$ ”; in category theory, we generalize this using a partial order on subobjects.

*The Ordering of Subobjects.* Let  $X$  be an object in  $\mathcal{C}$ . We define a relationship ( $\leq$ ) on monomorphisms into  $X$  as follows:

1. **The Preorder:**  $u : A \rightarrow X \leq v : B \rightarrow X$  if there exists a morphism  $w : A \rightarrow B$  such that  $v \circ w = u$ . This relation defines a preorder.
2. **From Preorder to Partial Order:** By grouping monomorphisms into  $X$  into isomorphism classes, this relation induces a partial order on the collection  $\text{Sub}_{\mathcal{C}}(X)$  of subobjects of  $X$ .

For any object  $X$ ,  $(\text{Sub}_{\mathcal{C}}(X), \leq)$  behaves like the power set  $\mathcal{P}(X)$  partially ordered by subset inclusion  $\subseteq$ .<sup>7</sup>

With our definitions of interpretation and subobject ordering in place, we can now formally define what it means for a logic statement to be “true” or *satisfied* in a given structure.

1. **Satisfaction of a Sequent.** Let  $M$  be a  $\Sigma$ -structure over an observable category  $\mathcal{C}$ . Consider an observable sequent  $\sigma := (\phi \vdash_{\Gamma} \psi)$  over  $\Sigma$ . We say that the structure  $M$  satisfies the sequent (written  $M \models \sigma$ ) if:

$$M\phi^{\Gamma} \leq M\psi^{\Gamma} \quad \text{in } \text{Sub}_{\mathcal{C}}(M\Gamma).$$

This inequality states that the “space” of assignments satisfying the premise  $\phi$  is entirely contained within the “space” of assignments satisfying the conclusion  $\psi$ . In the language of category theory, this means there exists a (necessarily unique) morphism from  $M\phi^{\Gamma}$  to  $M\psi^{\Gamma}$  in the full subcategory of the slice category  $\mathcal{C}/M\Gamma$  whose objects are the monomorphisms into  $M\Gamma$ .

In Lean, this is implemented as a family of propositions indexed by sequents.

```
def Sequent.IsSatisfiedIn (seq : σ.Sequent) : Prop :=
  Nonempty (seq.left.intrpIn M seq.isFormula_left →
    seq.right.intrpIn M seq.isFormula_right)
```

<sup>7</sup> By the definition of an observable category, the subobject poset  $\text{Sub}_{\mathcal{C}}(X)$  is a lattice with arbitrary unions stable under pullback for every object  $X$  of  $\mathcal{C}$ . This stability property allows substitution to distribute over unions.

2. **Models of a Theory.** Finally, we extend this to the level of an entire observable theory  $\mathbb{T}$  (a set of axioms) over  $\Sigma$ . We say that  $M$  is a model of  $\mathbb{T}$  (or a  $\mathbb{T}$ -model) if every axiom in the theory is satisfied in  $M$ . We denote this as  $M \models \mathbb{T}$ .

In Lean, this is implemented as a family of propositions indexed by observable theories.

```
def Str.IsModelOf (T :  $\sigma$ .Theory) : Prop :=  $\forall$  seq  $\in$  T.axioms, seq.IsSatisfiedIn M
```

### 2.3 Soundness: Bridging Syntax and Semantics

The previous sections established two distinct worlds: the syntactic world of formal rules and provability ( $\vdash$ ), and the semantic world of models and satisfaction ( $\models$ ). We now connect these two domains through a fundamental metaproperty: **soundness**.

Soundness provides the guarantee that our deduction system is “correct”. Formally, it states:

If a sequent  $\sigma$  is provable in a theory  $\mathbb{T}$  ( $\mathbb{T} \vdash \sigma$ ), then  $\sigma$  is satisfied in every  $\mathbb{T}$ -model  $M$  ( $M \models \mathbb{T} \implies M \models \sigma$ ).

**Why Soundness Matters for AlphaGeometry.** In the context of state-of-the-art geometry solvers like AlphaGeometry (AG), reasoning is often performed by a combination of neural heuristics and symbolic engines. To the best of our knowledge, the current underlying reasoning engine of AG (the DD+AR system) has not been proven sound. This leaves a trust gap in AG: there is no formal guarantee that the system’s deductions always hold true. This gap calls for a human expert review, not only for an overall evaluation or grading of AG’s solutions, but possibly for their correctness.

In this work, we provide a future generalization of AlphaGeometry, based on the implementation of the logic of observation in Lean, with a complete mechanized proof of soundness for its deduction system. This proof is not restricted to geometry; rather, it covers the entire class of observable theories in one fell swoop. As previously noted, this includes:

- Euclidean and affine geometries.
- Algebraic theories (groups, rings, etc.).
- Abstract theories (small categories, graphs).

Proving soundness at this level of abstraction ensures generality, while mechanizing this theory-agnostic proof in Lean removes the possibility of subtle flaws in our implementation, such as errors in variable binding or substitution.

**The Soundness Theorem.** *Let  $\mathbb{T}$  be an observable theory over a signature  $\Sigma$ ,  $M$  be a model of  $\mathbb{T}$  in an observable category, and let  $\sigma$  be an observable sequent over  $\Sigma$ . If  $\mathbb{T} \vdash \sigma$ , then  $M \models \sigma$ . [15, Prop. D1.3.2]*

*Proof.* By structural induction on the proof tree of  $\mathbb{T} \vdash \sigma$ .

Idea: We demonstrate that if the premises of a rule in Appendix A are satisfied in  $M$ , then the conclusion is likewise satisfied. This is where the observable nature of the category  $\mathcal{C}$  is vital; we utilize the categorical properties of intersections (for  $\wedge$ ), unions (for  $\vee$ ), and images (for  $\exists$ ) to mirror the logical operations, e.g. moving variables in and out of scope is handled by projections ( $p$ ) and pullbacks ( $p^*$ ), and the double rule for existential quantification for instance is the standard fact that the image functor in an observable category is left adjoint to the pullback functor. Crucially, the proof relies on a technical lemma characterizing the interpretation of a formula  $\phi$  under a substitution between two contexts [15, Lem. D1.2.7]:

**Lemma 1.** *Let  $\gamma : \text{Subst}(\Gamma, \Delta)$  be a substitution between contexts  $\Gamma$  and  $\Delta$ , and let  $\Gamma \vdash \phi$  be a formula in context  $\Gamma$ . Then there exists a pullback square*

$$\begin{array}{ccc} M(\phi[\gamma])^\Delta & \xrightarrow{\quad} & M\phi^\Gamma \\ \downarrow & & \downarrow \\ M\Delta & \xrightarrow{M\gamma} & M\Gamma. \end{array}$$

In Lean, we establish this property by induction on the formula  $\phi$ , formalising the statement as follows.

```
lemma substitution : ∀ (φ : σ.Formula) {Γ Δ : σ.Ctx} (p : Γ ⊢ φ) (γ : Γ.Subst Δ),
  Nonempty ((MonoOver.pullback (γ.intrpIn M)).obj (φ.intrpIn M p) ≡
    (φ[γ] p).intrpIn M (γ.formula_inCtx p))
```

This lemma is itself an extension of the corresponding substitution property for terms [15, Lem. D1.2.4], which is proved similarly by induction.

**Lemma 2.** *Let  $\gamma : \text{Subst}(\Gamma, \Delta)$  be a substitution between contexts  $\Gamma$  and  $\Delta$ , and let  $\Gamma \vdash t : S$  be a term of type  $S$  in context  $\Gamma$ . Then the interpretation  $M(t[\gamma])^\Delta$  is the composite*

$$M\Delta \xrightarrow{M\gamma} M\Gamma \xrightarrow{Mt^\Gamma} MS.$$

```
lemma Term.substitution : ∀ (t : σ.Term s) (p : Γ ⊢ t) (γ : Γ.Subst Δ),
  (t[γ] p).intrpIn M (γ.term_inCtx p) = γ.intrpIn M >> t.intrpIn M p
```

*Implementation Note.* To ensure the Lean formalization remains manageable and clean, we encapsulated the core inductive logic within an auxiliary lemma, `provable_sequent_isSatisfied`. This lemma comprises 648 lines of code, inclusive of the substitution lemmas and their respective proofs. This auxiliary lemma performs the heavy lifting by executing the structural induction over the proof tree, enabling a concise and high-level proof of the main Soundness Theorem.

```
theorem soundness :
  ∀ (seq : σ.Sequent), M.IsModelOf T → T.Provable seq → seq.IsSatisfiedIn M := by
```

```

intros seq m h
rcases h with (π)
unfold IsSatisfiedIn
apply provable_sequent_isSatisfied
exact π
exact m

```

**Soundness-as-a-Service for Specialized Solvers.** Crucially, our implementation of the Soundness Theorem is fully parametrized by an observable theory  $\mathbb{T}$  and its underlying signature  $\Sigma$ . As a result, this Soundness Theorem is not a static result locked into Euclidean geometry. Rather, we have developed a plug-and-play architecture for formal verification. This means that for a team using our framework, proving that a new system –such as a specialized solver for affine geometry– is sound becomes a matter of simply providing the corresponding signature and the theory’s axioms; the framework’s core soundness result then applies automatically, significantly reducing the burden of formal verification.

## 2.4 Two Illustrations: Affine and Euclidean Geometries

AlphaGeometry focuses on synthetic plane geometry. This focus is driven by the need to solve the most challenging, creative problems in classical Euclidean geometry, a task where traditional AI methods struggle with the sheer complexity and lack of intuition. This complexity stems from the requirement for auxiliary constructions: many of the most difficult problems in synthetic geometry require creative auxiliary constructions to unlock the solution, such as adding a new point, line, or circle to a diagram. This is where a key weakness of traditional geometric solvers lies. AG’s neuro-symbolic approach uses a neural language model to generate these creative, human-like constructions and guide a symbolic deduction engine, mimicking the way human experts think. IMO-style geometry problems are generally posed and solved in the synthetic tradition, and AG uses IMO-level benchmarks as a compelling frontier in AI mathematical reasoning.

**Domain Specificity and Methodological Constraints in AlphaGeometry.** AG was engineered to address the high-reasoning demands of IMO geometry, achieving performance parity with gold-medalists through its neuro-symbolic framework (see [6]). A core innovation of AG is its synthetic data-generation pipeline, which bypasses human-designed problem sets to sample theorem premises randomly. As noted by the authors, this process avoids “human aesthetic biases”, such as a preference for symmetry, thereby exploring a broader set of scenarios than traditional human-curated curriculums [22]. However, while AG successfully mitigates aesthetic bias, it restricts itself to synthetic geometry. While AG’s methodology for generating synthetic data and dealing with auxiliary constructions (the creative leaps typical of Euclidean proofs) is

broadly applicable across mathematical theories, its domain-specific language and its optimized deduction engine currently limit AG’s ability to generalize beyond synthetic plane geometry. In contrast, analytic geometry for instance relies on the language of coordinate systems and on extensive algebraic manipulation. Moreover, analytic frameworks present a different computational challenge that requires more general symbolic engines to meet their demands with the same level of efficiency.

**Affine and Euclidean Geometries.** To illustrate how a future iteration of AlphaGeometry, based on the framework we have described, would cover a range of theories, and to ground the abstract theory in concrete examples, we illustrate how two mathematical domains –in this case, affine and Euclidean geometries– can be treated as two facets of a single symbolic polyhedron: they are both encoded as signatures within this common framework. Moreover, this provides a template for instantiating the *Signature* 2.1 structure in Lean.

*Affine Geometry.* Affine geometry can be viewed as an observable theory (as developed in work such as [5]). To represent it, we define a signature that captures the fundamental “types” of objects and the relationships between them.

Our signature  $\Sigma_{\text{Affine}}$  consists of the following components:

1. **Sorts:** We define two primary sorts  $\mathcal{P}$  and  $\mathcal{L}$  for points and lines, respectively.
2. **Function Symbols:** We should have three specific points that form our affine coordinate system, representing the origin  $O$  and the basis points  $I$ ,  $J$  of the plane. We model them as 0-ary function symbols (i.e. constants) with an empty input list of sorts and  $\mathcal{P}$  as the output sort.
3. **Relation Symbols:**  $\text{belong}(\mathcal{P}, \mathcal{L})$ , representing that a point lies on a specific line (incidence), and  $\text{parallel}(\mathcal{L}, \mathcal{L})$ , representing that two lines are parallel to each other.

*From Geometry to Lean.* This signature acts as the domain-specific language for our theory. By instantiating the *Signature* structure with these sorts and symbols, we create a formal language where we can then write the axioms of affine geometry.

```
inductive Sorts where
  | P : Sorts
  | L : Sorts
```

```
inductive Fun where
  | O : Fun
  | I : Fun
  | J : Fun
```

```
inductive Rel where
```

```

| belong : Rel
| parallel : Rel
| nonParallel : Rel
| distinctPoints : Rel
| distinctLines : Rel

notation "∈'" => Signature.AffGeom.Rel.belong
notation "||'" => Signature.AffGeom.Rel.parallel
notation "⋈'" => Signature.AffGeom.Rel.nonParallel
notation "≠p" => Signature.AffGeom.Rel.distinctPoints
notation "≠l" => Signature.AffGeom.Rel.distinctLines

def instSignature : Signature where
  Base := ℕ
  Index := ℕ
  leftIndex := 0
  rightIndex := 1
  Sorts := Signature.AffGeom.Sorts
  newLabel X := (X.sup Prod.fst + 1, by
    intro v hv
    have := Finset.le_sup (f := Prod.fst) hv
    exact ne_of_gt (Nat.lt_succ_of_le (n := v.1) (m := X.sup Prod.fst) this))
  Fun := Signature.AffGeom.Fun
  Rel := Signature.AffGeom.Rel
  rankF f := match f with
    | .0 => ([], .P)
    | .I => ([], .P)
    | .J => ([], .P)
  rankR R := match R with
    | ∈' => [.P, .L]
    | || => [.L, .L]
    | ⋈ => [.L, .L]
    | ≠p => [.P, .P]
    | ≠l => [.L, .L]

```

*Euclidean Geometry in the Observable Fragment.* To demonstrate the practical steps needed to move AlphaGeometry into our framework, we consider Euclidean geometry following the Tarskian axiomatization. This theory can be presented using a signature with a single sort for points, no function symbols, and two primary relation symbols:

1. Betweenness ( $B$ ): A ternary relation where  $B(a, b, c)$  denotes that point  $b$  lies on the segment between  $a$  and  $c$  (implying collinearity).
2. Congruence ( $C$ ): A 4-ary relation, denoted  $ab \equiv cd$  for simplicity, meaning the segment  $ab$  is congruent to the segment  $cd$ .

Because negation ( $\neg$ ) is not a primitive operator in the logic of observation (reflecting that the absence of a property is not always directly observable, e.g., if it requires to review an infinite number of cases) we must embed the theory

within the observable fragment of full first-order logic by explicitly introducing new relation symbols. Specifically, we add:

3. Apartness ( $a \neq b$ ): A binary relation denoting that  $x$  and  $y$  are distinct points.
4. Non-betweenness ( $\bar{B}(a, b, c)$ ): A ternary relation denoting that  $b$  is not between  $a$  and  $c$ .

By treating “apartness” and “non-betweenness” as primary observable relations rather than negations, we ensure that the theory of Euclidean geometry remains entirely within the observable fragment.

Tarski’s axioms for ruler-and-compass constructions can then be formulated as follows [20] (omitting explicit variable contexts for legibility):

1. Equidistance axiom E1:  $\top \vdash ab \equiv ba$
2. Equidistance axiom E2:  $(ab \equiv pq) \wedge (ab \equiv rs) \vdash pq \equiv rs$
3. Equidistance axiom E3:  $ab \equiv cc \vdash a = b$
4. Betweenness axiom:  $B(a, b, d) \wedge B(b, c, d) \vdash B(a, b, c)$
5. Segment Construction:  $\top \vdash (\exists x)(B(q, a, x) \wedge (ax \equiv bc))$
6. Five-Segment Axiom:  $(a \neq b) \wedge B(a, b, c) \wedge B(p, q, r) \wedge (ab \equiv pq) \wedge (bc \equiv qr) \wedge (ad \equiv ps) \wedge (bd \equiv qs) \vdash cd \equiv rs$
7. Pash Axiom:  $B(a, p, c) \wedge B(q, c, b) \vdash (\exists x)(B(a, x, q) \wedge B(b, p, x))$
8. Lower Dimension:  $\top \vdash (\exists a, b, c)[\bar{B}(a, b, c) \wedge \bar{B}(c, a, b) \wedge \bar{B}(b, c, a)]$
9. Upper Dimension:  $(a \neq b) \wedge (pa \equiv pb) \wedge (qa \equiv qb) \wedge (ra \equiv rb) \vdash [B(p, q, r) \vee B(r, p, q) \vee B(q, r, p)]$
10. Euclid’s Axiom:  $B(a, d, t) \wedge B(b, d, c) \wedge (a \neq d) \vdash (\exists x, y)[B(a, b, x) \wedge B(a, c, y) \wedge B(y, t, x)]$
11. Intersection Axiom:  $(ax \equiv ax') \wedge (az \equiv az') \wedge B(a, x, z) \wedge B(x, y, z) \vdash (\exists y')[(ay \equiv ay') \wedge B(x', y', z')]$ .

We also explicitly define the behavior of our additional relation symbols (Apartness and Non-betweenness):

12.  $\top \vdash (a = b) \vee (a \neq b)$
13.  $(a = b) \wedge (a \neq b) \vdash \perp$ ,

and similarly for  $\bar{B}$ .

*High-Level Abstractions and Diagrammatic Reasoning.* Importantly, there is no need to restrict ourselves to point-based signatures to formalize Euclidean geometry in our framework. It is possible to extend the signature above with additional primitive sorts –such as lines, circles, segments, angles, and areas– and treat them as first-class citizens by adding along them their corresponding relations, e.g. for representing various intersections. This extension allows the theory to form more high-level diagrammatic assertions, mirroring the way mathematicians (and current neuro-symbolic solvers) actually reason about geometry (see [2]). As long as we continue to introduce derived relations to handle negations (e.g., non-collinearity), these additional abstractions layers remain entirely within the logic of observation.

### 3 Conclusions and Future Work

In this paper, we have proposed a mathematically grounded language, the logic of observation, as a viable candidate for a cross-domain generalization of AlphaGeometry’s domain-specific language. By providing an implementation in Lean, we have moved one step closer to a unifying substrate for specialized systems such as AlphaGeometry. We have explicitly demonstrated how to embed Euclidean geometry and affine geometry within our framework. While our current implementation serves as a prototype, we acknowledge that it is not necessarily the definitive iteration; future refinements, such as the adoption of De Bruijn indexing for more elegant substitution handling, could further optimize the system’s implementation.<sup>8</sup>

Beyond the scalable language itself, we have implemented a complete set of inference rules, enabling the system to reason across a diverse range of theories. The primary advantage of this unified approach is the ability to prove meta-properties of the system once and for all; to that end, we have provided a machine-checked proof of soundness for our logic. This ensures that any AlphaGeometry-inspired engine built upon this foundation would be a formally verified deductive agent. By decoupling the high-level proof of soundness from the specific details of the domain (a specific theory over a given signature), we provide the logical substrate necessary for the bottom-right quadrant of the matrix in our introduction: a system that is as specialized as AlphaGeometry but as formally grounded and generalizable as AlphaProof.

We emphasize that this framework is not limited to the observable fragment; it possesses the inherent flexibility to be generalized beyond the logic of observation to full first-order theories and higher-order signatures. Recognizing AlphaGeometry, not as a specific program, but as an instance of a more general class of programs, we have a chance to move from a hard-coded geometry solver to a generalized reasoning architecture. In a sequel to this paper [4], the first author demonstrates how this level of generality could allow us to peer into the latent spaces of models like AlphaGeometry and provide a unique vantage point for mechanistic interpretability.

**Related Work** The authors are not aware of a directly related line of research trying to tackle the logical foundations of AlphaGeometry in order to extend its language towards a cross-domain system for mathematical reasoning. However, we can situate our work at the confluence of three established research areas:

*Formalization of First-Order Logic.* The formalization of first-order logic (FOL) is a well-established area in the proof assistant community. We note the rigorous formalization of single-sorted FOL with finitary disjunctions in Lean by [12]. Other significant efforts include the work of [16] in Coq; as well as the completeness and compactness developments in Isabelle’s Archive of Formal Proofs [11,21], the latter of which builds

---

<sup>8</sup> The decision to prioritize named variables over De Bruijn indexing in this version was made to ensure accessibility for project stakeholders with varying degrees of computer science knowledge.

upon Harrison’s foundational work in HOL Light [13]; and finally [1], a formalization of first-order logic in the PVS proof assistant to verify properties of unification algorithms and term-rewriting systems.

Our implementation focuses specifically on the many-sorted observable fragment –the logic of observation– and its contextual deduction engine. Unlike general-purpose FOL formalizations that prioritize completeness theorems for classical logic, our framework is optimized for its categorical semantics (see Section 2.2) as it will be explained in a forthcoming sequel to this article.

*Formalization of Euclidean Geometry.* In addition to the foundational work [20] cited in Section 2.4, we can also mention several key efforts across proof assistants: GeoCoq, a comprehensive formalization of geometry in Coq based on Tarski’s axiom system [3]; the work of [8] in Isabelle; and LeanGeo [19] in Lean, which is based on the diagrammatic system System E [2].

*Automated Theorem Proving for Geometry.* This field has transitioned from classical methods, such as Wu’s method [24] and the Area Method [7], to the modern neuro-symbolic approach pioneered by AlphaGeometry [22,6]. Recent works in the field include [18,25,9].

**Author Contributions.** A.B. conceived, initiated, and led this project on the logical foundations of AlphaGeometry, drawing on expertise in AI and formal verification. A.B. and F.J. collaborated on the Lean 4 formalization of the syntax of the logic of observation. F.J. developed the formalization of the semantics and the machine-checked proof of soundness. A.B. wrote the original manuscript. Both authors contributed to the review of the final text.

**Acknowledgments.** We thank all contributors to mathlib, both large and small, particularly the mathlib maintainers for their tireless work.

**Disclosure of Interests.** The authors have no competing interests to declare that are relevant to the content of this article.

## References

1. Avelar, A.B., Galdino, A.L., de Moura, F.L.C., Ayala-Rincón, M.: First-order unification in the pvs proof assistant. *Logic Journal of the IGPL* **22**(5), 758–789 (2014). <https://doi.org/10.1093/jigpal/jzu012>
2. Avigad, J., Dean, E., Mumma, J.: A formal system for euclid’s elements. *The Review of Symbolic Logic* **2**(4), 700–768 (2009)
3. Beeson, M., Boutry, P., Braun, G., Gries, C., Narboux, J.: *Geocoq* (2018)
4. Bordg, A.: The topological dual of a dataset: A logic-to-topology encoding for alphageometry-style data (Mar 2026). <https://doi.org/10.5281/zenodo.18961053>
5. Cataldi-Bamri, R., Pivet, E.: *Topos classifiants et géométrie affine*. Master’s thesis, Université Paris-Saclay (2023), available at <https://doi.org/10.6084/m9.figshare.23596656.v1>
6. Chervonyi, Y., Trinh, T.H., Olšák, M., Yang, X., Nguyen, H., Menegali, M., Jung, J., Verma, V., Le, Q.V., Luong, T.: Gold-medalist performance in solving olympiad geometry with alphageometry2. *arXiv preprint arXiv:2502.03544* (2025)

7. Chou, S.C., Gao, X.S., Zhang, J.: Machine proofs in geometry: Automated production of readable proofs for geometry theorems, vol. 6. World Scientific (1994)
8. Coghetto, R.: Tarski’s parallel postulate implies the 5th postulate of euclid, the postulate of playfair and the original parallel postulate of euclid. *Archive of Formal Proofs* (January 2021), <https://isa-afp.org/entries/IsaGeoCoq.html>, Formal proof development
9. Duan, B., Liang, X., Lu, S., Wang, Y., Shen, Y., Chang, K.W., Wu, Y.N., Yang, M., Chen, W., Gong, Y.: Gold-medal-level olympiad geometry solving with efficient heuristic auxiliary constructions (2025), <https://arxiv.org/abs/2512.00097>
10. Fiore, M., Plotkin, G., Turi, D.: Abstract syntax and variable binding. In: *Proceedings. 14th Symposium on Logic in Computer Science* (Cat. No. PR00158). pp. 193–202. IEEE (1999)
11. From, A.H.: Soundness and completeness of an axiomatic system for first-order logic. *Archive of Formal Proofs* (September 2021), [https://isa-afp.org/entries/FOL\\_Axiomatic.html](https://isa-afp.org/entries/FOL_Axiomatic.html), Formal proof development
12. Han, J.M., van Doorn, F.: A formal proof of the independence of the continuum hypothesis. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. pp. 353–366 (2020)
13. Harrison, J.: Formalizing basic first order model theory. In: *International Conference on Theorem Proving in Higher Order Logics*. pp. 153–170. Springer (1998)
14. Hubert, T., Mehta, R., Sartran, L., Horváth, M.Z., Žužić, G., Wieser, E., Huang, A., Schrittwieser, J., Schroecker, Y., Masoom, H., et al.: Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature* pp. 1–3 (2025)
15. Johnstone, P.T.: *Sketches of an Elephant: A Topos Theory Compendium: Volume 2*, vol. 2. Oxford University Press (2002)
16. Kirst, D., Hostert, J., Dudenhefner, A., Forster, Y., Hermes, M., Koch, M., Larchey-Wendling, D., Mück, N., Peters, B., Smolka, G., et al.: A coq library for mechanised first-order logic. In: *The Coq Workshop 2022* (2022)
17. Moura, L.d., Ullrich, S.: The lean 4 theorem prover and programming language. In: Platzner, A., Sutcliffe, G. (eds.) *Automated Deduction – CADE 28*. pp. 625–635. Springer International Publishing, Cham (2021)
18. Sicca, V., Xia, T., Fédérico, M., Gorinski, P.J., Frieder, S., Jui, S.: Newclid: A user-friendly replacement for alphageometry with agentic support. *arXiv preprint* (2024)
19. Song, C., Wang, Z., Pu, F., Wang, H., Lin, X., Liu, J., Li, J., Liu, Z.: Leangeo: Formalizing competition geometry problems in lean. *arXiv preprint arXiv:2508.14644* (2025)
20. Tarski, A., Givant, S.: Tarski’s system of geometry. *Bulletin of Symbolic Logic* **5**(2), 175–214 (1999)
21. Tourret, S., Paulson, L.C.: Compactness theorem for first-order logic. *Archive of Formal Proofs* (February 2025), [https://isa-afp.org/entries/FOL\\_Compactness.html](https://isa-afp.org/entries/FOL_Compactness.html), Formal proof development
22. Trinh, T.H., Wu, Y., Le, Q.V., He, H., Luong, T.: Solving olympiad geometry without human demonstrations. *Nature* **625**(7995), 476–482 (2024)
23. Vickers, S.: *Topology via logic*. Cambridge University Press (1989)
24. Wu, W.t.: On the decision problem and the mechanization of theorem-proving in elementary geometry. In: *Selected Works Of Wen-Tsun Wu*, pp. 117–138. World Scientific (2008)
25. Zhao, H., Shen, J., Zhang, Y., Gao, S., Liu, K., Ma, T., Zheng, F., Lin, D., Zhang, W., Chen, K.: Achieving olympia-level geometry large language model agent via

complexity boosting reinforcement learning (2025), <https://arxiv.org/abs/2512.10534>

## A Deduction System

The structural rules consist of the *identity axiom*

$$\phi \vdash_{\vec{x}} \phi,$$

the *substitution rule*

$$\frac{\phi \vdash_{\vec{x}} \psi}{\phi[\vec{t}/\vec{x}] \vdash_{\vec{y}} \psi[\vec{t}/\vec{x}]}$$

where  $\vec{y}$  is any string of variables including all the variables occurring in the string of terms  $\vec{t}$ , and the *cut rule*

$$\frac{(\phi \vdash_{\vec{x}} \psi) \quad (\psi \vdash_{\vec{x}} \chi)}{\phi \vdash_{\vec{x}} \chi}.$$

The rules for logical connectives consist of the rules for *finite conjunction*, which are the axioms

$$\phi \vdash_{\vec{x}} \top, \quad \phi \wedge \psi \vdash_{\vec{x}} \phi, \quad \phi \wedge \psi \vdash_{\vec{x}} \psi,$$

and the rule

$$\frac{(\phi \vdash_{\vec{x}} \psi) \quad (\phi \vdash_{\vec{x}} \chi)}{\phi \vdash_{\vec{x}} \psi \wedge \chi};$$

the rules for *infinitary disjunction*, which are the axioms

$$\perp \vdash_{\vec{x}} \phi, \quad \phi_{i_0} \vdash_{\vec{x}} \bigvee_{i \in I} \phi_i,$$

and the rule

$$\frac{\phi_i \vdash_{\vec{x}} \psi \text{ for every } i \in I}{\bigvee_{i \in I} \phi_i \vdash_{\vec{x}} \psi},$$

for every  $i_0 \in I$  and where  $I$  is any, possibly infinite, set; the double rule for *existential quantification*

$$\frac{\phi \vdash_{\vec{x}, y} \psi}{(\exists y)\phi \vdash_{\vec{x}} \psi}$$

(assuming  $y$  is not free in  $\psi$ ); and the “mixed rules”, namely the *distributivity axiom*

$$\phi \wedge \left( \bigvee_{i \in I} \psi_i \right) \vdash_x \bigvee_{i \in I} (\phi \wedge \psi_i)$$

and the *Frobenius axiom*

$$\phi \wedge (\exists y)\psi \vdash_{\vec{x}} (\exists y)(\phi \wedge \psi),$$

where  $y$  is not in the context  $\vec{x}$  (hence, is not free in  $\phi$ ).

Last, the *equality rules* consist of the axioms

$$\top \vdash_x x = x$$

and

$$(\vec{x} = \vec{y}) \wedge \phi \vdash_{\vec{z}} \phi[\vec{y}/\vec{x}],$$

where  $\vec{x}$  and  $\vec{y}$  are contexts of the same length and type,  $\vec{x} = \vec{y}$  is a shorthand for the conjunction  $(x_1 = y_1) \wedge \cdots \wedge (x_n = y_n)$ , and  $\vec{z}$  is any context containing  $\vec{x}$ ,  $\vec{y}$  and the free variables of  $\phi$ . (The symmetry and transitivity of equality can then be deduced from the previous rules.)