First-Order Logic: Syntax and Semantics

Alan Fern, afern@eecs.oregonstate.edu

March 6, 2020

We now know the syntax and semantics of first-order logic. However, a logic is only as useful as the conclusions we can deduce from it. Here we will discuss approaches to first-order deductive inference. The inference algorithms we consider can all be viewed as "first-order upgrades" of the propositional algorithms. The process of creating such an upgrade is often referred to **lifting**. As a side note, similar, but usually simpler, lifting ideas are often used in moving from propositional (or fixed size) representations in machine learning to representations with variable numbers of objects and relations.

Below we will define several subclasses of first-order (FO) logic, discuss inference techniques for these classes, as well as inference for full first-order logic. Finally, we will cover the basic idea behind Göedel's incompleteness theorem which highlights some fundamental limitations of FO logic. These notes, are not complete, and are intended to supplement the textbook.

1 Types of FO Theories

A FO Theory is just a set of FO formulas, which we generally think of a being a conjunction of those formulas. Intuitively, a theory aims to capture all of the constraints about a system or world that are necessary for reasoning about that system or world.

For computational reasons it is common to develop algorithms for particular subclasses of FO logic, where the subclasses are defined by placing syntactic restrictions on the allowed formulas. In this Section, we define several of these classes.

1.1 FO Ground Theories

Recall that a FO **ground formula** is a FO formula that does not contain variables. A **FO ground theory** is simply a set of ground formulas. We will see below that FO ground theories are essentially syntactic sugar for propositional logic. That is, one can reason about ground theories using purely propositional methods.

1.2 FO Clausal Forms

A clause is defined as follows.

- An **atom** is simply a predicate symbol applied to the appropriate number of terms, possibly where the terms contain variables, such as, $TallerThan(Bob, FatherOf(Bob))$, or $TallerThan(x, y)$.
- A **literal** is either an atom or the negation of an atom.

- A **clause** is a disjunction of literals where all variables are universally quantified.

An example of a clause is

$$\forall x, y, z \neg TallerThan(x, y) \vee \neg TallerThan(y, z) \vee TallerThan(x, z)$$

Typically, we do not actually write the universal quantifier, so the above would just be written as

$$\neg TallerThan(x, y) \vee \neg TallerThan(y, z) \vee TallerThan(x, z)$$

It is convention to interpret any free variables as universally quantified.

1.2.1 FO Definite Clausal Theory

Definite clauses are clauses that have EXACTLY ONE positive literal.

The above clause is a definite clause.

A definite clause is equivalent to an implication whose antecedent (or body) is the conjunction of negative literals and the consequent (or head) is the single positive literal. So the above definite clause is equivalent to the more intuitive,

$$(TallerThan(x, y) \wedge TallerThan(y, z)) \Rightarrow TallerThan(x, z)$$

which constrain $TallerThan$ to be a transitive relation.

Definite clausal theories are sets of definite clauses.

An example of a definite clausal theory might be,

$$\begin{aligned} TallerThan(x, y) \wedge TallerThan(y, z) \Rightarrow TallerThan(x, z) \\ TallerThan(Bob, FatherOf(Bob)) \\ TallerThan(FatherOf(Bob), FatherOf(FatherOf(Bob))) \\ TallerThan(Jon, x) \end{aligned}$$

The second and third clauses are called ground **unit clauses** (or ground atoms) and are best thought of as stating facts in a database. The fourth unit clause contains a variable x. Recalling that x is universally quantified, the clause should be interpreted as stating that Jon is taller than all other objects (including himself, so $TallerThan$ is not a strict "inequality"). It should be obvious that the atom $TallerThan(Bob, FatherOf(FatherOf(Bob)))$ is entailed by this theory.

Definite clausal theories are important because for many applications they provide enough expressive power to get the job done, and there are a number of highly optimized approaches for inference with such theories. Also definite clauses are quite intuitive to write down as they can be thought of as "if/then" rules. That is, if the body is satisfied by the facts that we know, then we can infer a new fact from the head. However, it is important to realize that definite clausal theories are strictly less expressive than full first-order logic. That is, there are FO formulas that can be written down, such that there is no logically equivalent definite clausal theory.

1.2.2 Horn Clauses

Horn clauses are clauses that contain AT MOST one positive literal.

Thus Horn clauses/theories are a superset of definite clauses/theories that allow for clauses with zero positive literals. We will not discuss Horn theories in this course, but they are an important subclass that are often used in practice, and share many of the computational advantages of definite theories. In particular, Horn clauses are the basis for the Prolog programming language, which falls in the declarative paradigm of programming languages. The purest form of Prolog program is simply a Horn theory.

1.2.3 Datalog Theories

Finally there is a distinguished subset of definite clausal theories that has nice computational properties.

Datalog theories are definite clausal theories that contains no function symbols.

The above theory is not a Datalog theory because the second and third clauses contains the $FatherOf$ function. Datalog is a popular language for database applications due to the nice computational properties. Intuitively, by not including function symbols the set of objects that can be referenced during inference is fixed to the set of constants/terms in the theory. For example, consider the non-Datalog clause $Num(x) \Rightarrow Num(Succ(x)$. If we know the ground atom $Num(0)$ is true, then we can infer that $Num(Succ(0))$ is true, showing that a new ground term $Succ(0)$ was "created" during inference. An unbounded number of new terms could be created in this way. This is not allowed in Datalog.

2 Propositional Inference for Ground Theories

Given a first-order ground theory KB, the follow steps can be used to create a corresponding propositional theory $\textsc{Prop}(KB)$:

1. Assign a unique propositional symbol to each ground atom that appears in KB,
2. Create a propositional theory by replacing each ground atom in KB with its assigned proposition.

For example, the ground theory

$$\begin{aligned} ThrillSeeker(Bob) \Rightarrow [Owns(Bob, Car) \wedge Fast(Car)] \\ ThrillSeeker(Bob) \end{aligned}$$

where Bob and Car are constants, corresponds to the propositional theory

$$\begin{gathered} P_1 \Rightarrow [P_2 \wedge P_3] \\ P_1 \end{gathered}$$

where P_1, P_2, and P_3 correspond to the first-order atoms $ThrillSeeker(Bob)$, $Owns(Bob, Car)$, and $Fast(Car)$. It is a simple matter to argue that a first-order ground theory KB is satisfiable (i.e. has a first-order model) iff $\textsc{Prop}(KB)$ is satisfiable (i.e. has a propositional model). You should convince yourself of this.

Given the equivalence between KB and $\textsc{Prop}(KB)$, one can perform inference for KB by applying propositional inference algorithms to $\textsc{Prop}(KB)$. This approach to inference is sound and complete as long as the propositional algorithm is sound and complete. Thus, in a strong sense, ground first-order theories are inherently propositional. The fact that they contain relations and structured terms serves no purpose other than providing a convenient way of naming propositions (syntactic sugar).

This shows, the perhaps obvious fact, that fundamentally first-order ground theories are no more expressive than propositional logic. FO logic gains its expressive power exclusively through the use of variables and quantifiers.

3 Substitutions and Unification

In order to "lift" propositional inference techniques to the case of non-ground first-order theories, we will utilize the concepts of variable substitution and unification.

3.1 Variable Substitution

A **variable substitution** θ is simply an ordered set (i.e. list) of pairs

$$\theta = \{v_1/t_1, \cdots, v_n/t_n\}$$

where each pair v_i/t_i contains a variable v_i along with a term t_i that will replace (be substituted for) v_i. Two example substitutions are

$$\theta_1 = \{x/Bob, y/FatherOf(Bob)\}$$

meaning replace x with Bob and then replace y with $FatherOf(Bob)$), and

$$\theta_2 = \{x/y, y/FatherOf(z)\}$$

meaning replace x with y, and then replace y with $FatherOf(z)$.

Given a formula ϕ and a substitution involving free variables in ϕ we define

$$\textsc{Subst}(\theta, \phi)$$

to be a new formula that results by starting with ϕ and then sequentially replacing the variables of ϕ with the corresponding terms in the substitution.

So for example if

$$\phi = TallerThan(x, y) \wedge TallerThan(y, z)$$

then

$$\textsc{Subst}(\theta_1, \phi) = TallerThan(Bob, FatherOf(Bob)) \wedge TallerThan(FatherOf(Bob), z)$$
$$\textsc{Subst}(\theta_2, \phi) = TallerThan(FatherOf(z), FatherOf(z)) \wedge TallerThan(FatherOf(z), z)$$

Sometimes substituting a term for a variable is referred to as **instantiating** the variable with the term.

3.2 Variable Unification

The second concept we will use for dealing with variables is unification. A **unification** for two formulas ϕ_1 and ϕ_2 is a substitution θ such that,

$$\textsc{Subst}(\theta, \phi_1) = \textsc{Subst}(\theta, \phi_2) \ .$$

That is, a unification is a substitution that makes two formulas syntactically equivalent.

Note that there can be many possible unifications between two formulas. For example, the formulas $P(x)$ and $P(f(y))$ have an infinite number of unifications including $\theta_1 = \{x/f(y)\}$ and $\theta_2 = \{x/f(f(z)), y/f(z)\}$. Notice that after applying θ_1 to these formulas resulting in $P(f(y))$ it is possible to apply another substitution $\theta_3 = \{y/f(z)\}$ in order to obtain the result of applying θ_2. More formally, given two formulas ϕ_1 and ϕ_2 we say that a unifier θ_1 is "more general" than θ_2, whenever a θ_3 exists such that,

$$\textsc{Subst}(\theta_3, \textsc{Subst}(\theta_1, \phi_1)) = \textsc{Subst}(\theta_2, \phi_1)$$

or equivalently

$$\textsc{Subst}(\theta_3, \textsc{Subst}(\theta_1, \phi_2)) = \textsc{Subst}(\theta_2, \phi_2)$$

Intuitively, this means that the set of formulas that can be obtained by substitutions after applying θ_1 are a superset of those that could be obtained by substitutions after applying θ_2. Notice that θ_2 is not more general than θ_1, since there is no substitution θ such that $\textsc{Subst}(\theta_3, P(f(f(z)))) = P(f(y))$.

In our inference algorithms we will only be interested in computing a **most general unifier** of two formulas. That is, a unifier such that no other unifier is more general. It turns out that there is always a most general unifier. Your book gives an algorithm for computing a most general unifier that makes ϕ_1 identical to ϕ_2. We will denote this unifier by $\textsc{Unify}(\phi_1, \phi_2)$.

4 Inference for Definite Clausal Theories

Given a definite clausal theory KB and a query atom α, our goal is to determine whether or not $KB \models \alpha$. Recalling that definite theories can be viewed as if/then rules with variables, it is relatively straightforward to construct forward and backward chaining techniques for inference. Both forward and backward chaining will make use of an inference rule known as **generalized modus ponens**:

$$\frac{a_1', a_2', \cdots, a_n',\ (a_1 \wedge a_2 \wedge \cdots \wedge a_n \Rightarrow q)}{\textsc{Subst}(\theta, q)}$$

where the a_i' and a_i are atoms such that

$$\forall i \ \ \textsc{Subst}(\theta, a_i') = \textsc{Subst}(\theta, a_i) \ .$$

So, for example, given the rule

$$TallerThan(x, y) \wedge TallerThan(y, z) \Rightarrow TallerThan(x, z)$$

along with the atoms

$$TallerThan(Jon, Bob) \text{ and } TallerThan(Bob, FatherOf(Bob)) \ ,$$

we can conclude $TallerThan(Jon, FatherOf(Bob))$, where θ here is

$$\{x/Jon, y/Bob, z/FatherOf(Bob)\} \ .$$

In other words, generalized modus ponens says that if we can find a substitution so that the premise of a rule (here $a_1 \wedge \cdots \wedge a_n$) is satisfied by a given set of facts (here $a'_1, \cdots, a'_n$) then we can derive the rule's conclusion under the substitution. *It turns out that for definite clausal theories, generalized modus ponens is the only inference rule we will require.*

4.1 Forward Chaining

We first consider **forward-chaining inference**, which uses modus ponens to iteratively enumerate the entailed facts of the definite theory.

Denote by F_i the set of facts that can be derived from the theory using i or fewer applications of generalized modus ponens. Forward chaining begins with the empty set of facts F_0 and on the i^{th} iteration computes $F_i = F_{i-1} \cup F'$, where F' is the set of all facts that can be derived via any application of generalized modus ponens to the facts in F_{i-1} and the rules in our theory. Your book gives more formal pseudo-code for this procedure. The iteration ends when F_i contains the query or $F_i = F_{i-1}$ (i.e. we reached a fixed point).

For example, given the KB

$$\begin{gathered}
TallerThan(x, y) \wedge TallerThan(y, z) \Rightarrow TallerThan(x, z) \\
TallerThan(Jon, Bob) \\
TallerThan(Bob, FatherOf(Bob)) \\
TallerThan(FatherOf(Bob), FatherOf(FatherOf(Bob)))
\end{gathered}$$

and the query $TallerThan(Jon, FatherOf(FatherOf(Bob)))$, forward chaining starts with $C_0 = \{\}$ and computes

$$\begin{aligned}
F_1 \quad = \quad \{ \ & TallerThan(Jon, Bob) \\
& TallerThan(Bob, FatherOf(Bob)) \\
& TallerThan(FatherOf(Bob), FatherOf(FatherOf(Bob))) \\
& TallerThan(Jon, FatherOf(Bob)) \\
& TallerThan(Bob, FatherOf(FatherOf(Bob))) \ \}
\end{aligned}$$

then given F_1 we see that we can derive $TallerThan(Jon, FatherOf(FatherOf(Bob))$ which would appear in F_2, ending the forward chaining process and proving that the query is entailed by the KB. The main complexity of forward chaining is in searching for possible rule applications on each iteration. Your book discusses a number of approaches for speeding up this process.

4.1.1 Forward Chaining: Datalog

As briefly mentioned in your book, forward chaining is sound and complete for Datalog theories. This is because for Datalog theories, the lack of function symbols allows us to put a finite bound on the number of possible ground terms (the only ground terms are constants) and hence the number of possible atoms that can be derived. Thus, forward chaining is guaranteed to terminate in a finite number of iterations at a fixed point. In addition, it can be shown that the fixed point contains exactly the entailed facts.

4.1.2 Forward Chaining: General Definite Theories

When we allow function symbols the story changes. Forward chaining may not always return an answer for general definite theories. To see this, note that when function symbols are allowed, there can be an infinite number of terms (by nesting of function symbols), yielding an infinite number of possible atoms. This means that forward chaining can possibly iterate forever, producing new facts and never reaching a finite fixed point.

For example, consider the theory,

$$\begin{gathered} TallerThan(Bob, FatherOf(Bob)) \\ TallerThan(x, FatherOf(x)) \Rightarrow TallerThan(FatherOf(x), FatherOf(FatherOf(x))) \end{gathered}$$

and the query $TallerThan(Bob, Bob)$. When we run forward chaining on this theory each iteration produces a new fact. In particular, iteration i produces the fact

$$TallerThan(FatherOf^{i}(x), FatherOf^{i+1}(x)) \; .$$

Since $TallerThan(Bob, Bob)$ is not entailed, it will never be generated by the forward chaining process; hence, the algorithm will never halt, and it will not answer the query.

This shows that forward chaining is not guaranteed to produce answers for queries that are not entailed by KB; however, it is guaranteed to correctly answer any query that is entailed, because forward chaining is guaranteed to eventually generate any atom that is entailed by the KB.

Thus, we see that forward chaining correctly handles entailed queries but may not correctly handle unentailed queries. This is not just a property of forward chaining. *Rather, it can be shown that, in general, for a definite clausal KB, the problem of deciding whether $KB \models \alpha$ is only semi-decidable.* This means that there are algorithms that can answer "yes" whenever $KB \models \alpha$, but there is no algorithm that can also correctly produce a "no" answer when $\neg(KB \models \alpha)$.

4.2 Backward Chaining of Definite Theories

Forward chaining can be viewed as searching for a sequence of modus ponens applications that derive the query. Naturally, we can also search for such a sequence backward from the query, which is known as **backward-chaining inference**.

Roughly speaking, backward chaining maintains a **goal set** that contains a set of facts that if proven true would imply the query. The goal set is initially set to be the query atom. Next, the procedure finds a rule/clause that allows for the query to be inferred, and then adds the antecedents to the goal set. This process of selecting a member of a goal set and adding antecedent facts continues until the goal set is empty (or – equivalently – when the goal set only contains facts that are in our KB). You should refer to your book for the details of this procedure.

As an example consider the following definite theory

$$\begin{aligned} \forall\, x\ (Person(x) \land Tall(x)) \Rightarrow TallerThan(x, Bob) \\ \forall\, x\, \forall\, y\ TallerThan(x, y) \Rightarrow Heavier(x, y) \\ Tall(Jon) \\ Person(Jon) \end{aligned}$$

and the query $Heavier(Jon, Bob)$.

Given the initial goal $Heavier(Jon, Bob)$, backward chaining will see that clause 2 is the only way to derive $Heavier(x, y)$ and will add a new goal $TallerThan(Jon, Bob)$ which is the antecedent of clause 2 after applying the substitution $\{x/Jon, y/Bob\}$. Likewise this goal can only be satisfied by clause 1 under the substitution $\{x/Jon\}$ and will result in a new goal set $\{Person(Jon), Tall(Jon)\}$. Both of these correspond to ground facts in the KB, indicating that we have successful proven/derived the query.

Your book has a more detailed treatment of backward chaining inference. Depending on the search strategy, backward chaining is complete for definite clausal theories (in the sense that any entailed query will be answered correctly). Backward chaining methods have been highly optimized and are the computational core of the `Prolog` programming language. `Prolog` defines programs using definite clausal theories, and it performs computation using backward chaining inference.

5 Resolution Inference for Full FO Logic

There are many first-order theories that can't be represented as definite clausal theories. In these cases, we need more powerful inference mechanisms than just forward and backward chaining.

First-order resolution is a widely used first-order inference method. The basic procedure is very simple. To decide whether $KB \models \alpha$, first convert the formula $KB \wedge \neg\alpha$ to first-order clausal form. Next, apply the resolution inference rule until arriving at a contradiction, which indicates that $KB \models \alpha$. We outline these steps below, noting that your book also gives some concrete examples.

5.1 Conversion to Clausal Form

Any well-formed formula of FO logic can be converted to an equivalent clausal theory—though not necessarily a definite clausal theory. This conversion is convenient since clauses then provide a uniform primitive representation upon which we can define inference rules and compute. The following steps can be applied to convert a formula to clausal form.

1. Eliminate all implication connectives, replacing them by disjunctions. So $P \Rightarrow Q$ becomes $\neg P \vee Q$, where P and Q are arbitrary FO logic (sub)formulas. This transformation preserves equivalence.

2. Push negation symbols "in" so that negations are only on individual atoms. This can be done using Demorgan's laws and also the first-order equivalents:

$$\neg\forall\, x\; \phi = \exists\, x\; \neg\phi \quad \text{and} \quad \neg\exists\, x\; \phi = \forall\, x\; \neg\phi$$

   This operation preserves equivalence.

3. Standardize variables apart. Rename all variables so that each quantifier has its own unique variable name. Obviously, this preserves equivalence.

4. Eliminate existential quantification (Skolemization). This is done by replacing existentially quantified variables by **Skolem constants** or **Skolem functions**. For example, convert $\exists x P(x)$ to $P(C)$ where C is a (Skolem) constant that does not appear anywhere in the original theory.

If the existential quantifier is within the scope of a universal quantified variable, the situation is a bit more complicated. For example, consider $\forall\ x\ \exists\ y\ P(x, y)$. It would not be correct to convert this to $\forall\ x\ P(x, C)$. This is because the resulting sentence says that the same object (represented by C) must satisfy $P(x, C)$ for all x. Rather the original sentence only requires that for each x there exist some object (possibly different for different x's) that satisfies $P(x, y)$. For this reason we will introduce a Skolem function, rather than constant. In this case we would get, $\forall\ x\ P(x, F(x))$ where F is a new function symbol that does not appear in the original theory.

As another example we would convert

$$\forall\, x\, \forall\, y\, \exists\, z\, P(x, y) \wedge P(y, z) \quad \text{to} \quad \forall\, x\, \forall\, y\, P(x, y) \wedge P(y, F(x, y))$$

here the Skolem function F must depend on both x and y since z was in their scope.

Intuitively we can think of existential quantifiers as stating that there exists some object with certain properties. Skolem constants and functions simply give concrete names for these objects that can be directly reasoned with. Since the Skolem symbols do not appear elsewhere in the theory, the only properties stated for these symbols are those that were stated for the existential variable, which is what we want.

Skolemization preserves equivalence relative to satisfiability. That is, the resulting theory is satisfiable iff the original theory was.

5. Remove universal quantifiers. This operation preserves equivalence given the previous operations. Recall that free variables in FO logic are defined to be universally quantified, so this is really just a formality.

6. Distribute $\wedge$ over $\vee$ to get a conjunction of disjunctions called **conjunctive normal form** (CNF). Convert $(P \wedge Q) \vee R$ to $(P \vee R) \wedge (Q \vee R)$, and convert $(P \vee Q) \vee R$ to $(P \vee Q \vee R)$. This gives a conjunction of clauses which continue to preserve equivalence.

7. Standardize variables in different clauses apart.

5.1.1 Example Conversion

Here is an example of the conversion process. Given the formula

$$\forall\, x\, \neg\, [\forall\, y\ (Q(x, y) \Rightarrow P(y))]$$

we first eliminate the implication

$$\forall\, x\, \neg\, [\forall\, y\ (\neg Q(x, y) \vee P(y))]\ .$$

Next, we push negation in

$$\begin{array}{l} \forall\, x\, \exists\, y\, \neg(\neg Q(x, y) \vee P(y)) \quad \text{followed by} \\ \forall\, x\, \exists\, y\ (Q(x, y) \wedge \neg P(y))\ . \end{array}$$

Then, we eliminate the existential quantifier by adding a Skolem function

$$\forall\, x\ (Q(x, F(x)) \wedge \neg P(F(x)))$$

and then drop universal quantifiers

$$Q(x, F(x)) \wedge \neg P(F(x)) \ .$$

The result is already in conjunctive normal form we just need to standardize apart variables in different clauses giving,

$$\begin{gathered} Q(x, F(x)) \\ \neg P(y, F(y)) \end{gathered}$$

5.2 First-Order Resolution Rule

The first-order resolution rule is just like the propositional version, only we use unification to match complementary literals in clauses, as follows.

$$\frac{P_1 \vee \cdots \vee P_n, \ Q_1 \vee \cdots \vee Q_m}{\textsc{Subst}\,(\theta, P_1 \vee ... \vee P_{j-1} \vee P_{j+1} \vee ... \vee P_n \vee Q_1 \vee ...Q_{k-1} \vee Q_{k+1} \vee ... \vee Q_m)}$$

where P_j and Q_k are such that $\textsc{Unify}(P_j, \neg Q_k) = \theta$.

Resolution proofs continue to apply the first-order resolution rule until reaching a contradiction (i.e. an empty clause).

5.2.1 FO Resolution Example

Given the two clauses

$$P(x) \vee Q(x) \quad \text{and} \quad \neg Q(y) \vee P(y) \ ,$$

we can quickly see that $P(T)$ is entailed by the two clauses for any term T. To see this we start with the formulas

$$P(x) \vee Q(x) \quad \text{and} \quad \neg Q(y) \vee P(y) \text{and} \ \neg P(T) \ ,$$

and wish to derive a contradiction.

First we see that there is a unifier for the complementary literals $Q(x)$ and $Q(y)$ in the two clauses:

$$\textsc{Unify}(Q(x), Q(y)) = \{x/y\} = \theta \ .$$

This allows us to apply the resolution rule, which produces the resolvent

$$\textsc{Subst}(\theta, P(x) \vee P(y)) = P(y) \vee P(y) = P(y) \ .$$

We can now trivially apply the resolution rule to $P(y)$ and and $\neg P(T)$ with the unifier $\{y/T\}$ which yields the empty clause, i.e. a contradiction. Note that the final step was really not necessary, since the first resolvent $P(y)$ is entailed by the original clauses (resolution is sound) and is equivalent to $\forall y \ P(y)$, which means we could directly derive $P(T)$ by just instantiating the quantifier for T. That is, resolution produced a direct proof rather than requiring the availability of $\neg P(T)$ to derive a contradiction.

The same idea applies to clauses with more complex terms; for example, the pair of clauses

$$P(f(f(x))) \vee Q(f(x)) \quad \text{and} \quad \neg Q(f(f(Bob)))$$

have

$$\textsc{Unify}(Q(f(x)), Q(f(f(Bob)))) = \{x/f(Bob)\}$$

and give the resolvent

$$\textsc{Subst}(\theta, P(f(f(x)))) = P(f(f(f(Bob)))) \ .$$

6 Completeness of Inference for First-Order Logic

First-order resolution is a refutation complete inference procedure! This is quite amazing given the simplicity of the rule and the expressiveness of FO logic. Note that this requires that unification is done using most general unifiers so that one doesn't "skip over" some consequences. Recall that refutation completeness means that resolution will produce a positive answer whenever $KB \wedge \neg\alpha$ is unsatisfiable (i.e. $KB \models \alpha$); however, first-order resolution may not terminate with an answer, in general, when $KB \wedge \neg\alpha$ is satisfiable (i.e. $KB \not\models \alpha$). There is no general stopping condition for resolution—we never really know if we are just 3 steps away from finding a contradiction, 1 million steps away, or whether we are on a never ending journey. We cannot expect any better, since, as stated above, even if we restrict ourselves to just definite clauses, the first-order entailment problem is only semi-decidable.

The resolution inference procedure was discovered and proven refutation complete in 1965 by John Robinson. Before that there were also complete proof systems (i.e. sets of inference rules) for full FO logic. The first proof that a proof system was complete was the *Completeness Theorem* of Kurt Gödel in 1929. At that time, it was an open problem of whether or not there was a complete proof procedure. Gödel's set of inference rules were larger and more complex than the single resolution rule, but it was also a fully complete procedure in the sense that if $KB \models \alpha$ it would be able to derive a finite proof of α starting only with KB. Since this soundness of the inference rules are trivial to show, it was possible to say for the first time that for any KB and α,

$$KB \models \alpha \iff KB \vdash \alpha.$$

Before that result the existence of a sound and complete proof procedure for FO logic was an important open problem.

7 Incompleteness of First-Order Logic

One of the great unsolved questions in mathematics in the early 1900s was whether or not mathematics could be fully mechanized. That is, could we develop mechanical procedures (i.e. algorithms) that could answer any mathematical question correctly in a finite amount of time. For example, is there an algorithm that can answer any question about number theory, such as "Is Fermat's last theorem true?" or "Is there a largest prime number?".

One way of addressing this question is via logic. Lets consider number theory as an important example domain. Any formal statement about natural numbers is either true or false in the standard model of natural numbers. We will call this model N. The domain of N is the natural numbers, the functions include at least the successor function, addition, and multiplication, and the relations include at least equality and $<$. Number theorists are interested in proving theorems that uncover interesting properties of the model N. It is important to reiterate that any statement about natural numbers is either true or false in this model. That is, for any FO model the interpretation of any formula is either true or false.

Suppose that we could write down a first-order theory KB of the natural numbers, such that for any statement α about numbers we have

$$KB \models \alpha \iff \alpha \text{ is true in } N \; .$$

From Kurt Gödel's completeness theorem we know that we can mechanically find a proof of α from KB. However, if α is not entailed then the mechanical procedure will not necessarily produce an answer. So how can we guarantee that we get an answer when α is not entailed? One answer is to wait for $\neg\alpha$ to be derived, which must be true under the above assumption, since if α is not true in N then $\neg\alpha$ must be true in N.

Another answer using resolution is to run two resolution proofs in parallel, one trying to prove α and the other trying to prove $\neg\alpha$. If KB satisfies the above property, then we know that – for any α – either $KB \models \alpha$ or $KB \models \neg\alpha$, and thus one of the resolution proofs is guaranteed to terminate in a finite amount of time, producing the correct answer.

So with the above argument, the question of whether number theory can be mechanized can be settled by finding a KB with the above property, showing that yes it can be mechanized. One of the greatest mathematical discoveries in the 20^{th} century was by Kurt Gödel, who gave a negative answer to this question with his *incompleteness theorem* in 1931.

The incompleteness theorem showed that for any KB sufficiently powerful to represent the basic axioms of number theory, there is always a formula α such that,

$$KB \not\models \alpha \quad \text{and} \quad KB \not\models \neg\alpha$$

This important result shows that no sound inference procedure will be able to derive either α or $\neg\alpha$, yet we know that one of these must be true in N. So, in a precise sense, it is not possible to write down a logical theory that completely captures the natural numbers—i.e. completely defines the model N. Rather any such attempt produces a KB that is true in N but also in some other unintended model N' that does not agree with N about all statements. If a formula is true in N but not in N', then it will not be entailed by KB and hence not provable. Adding further axioms to KB will not solve this problem, there will always be such unintended models.

The details and further consequences of this result can be found in any introductory textbook on mathematical logic. One of the most interesting consequences is that for any sufficiently powerful theory KB, it is not possible to prove the consistency of KB using just the axioms of KB. That is, we can't prove that KB doesn't entail contradictions. Intuitively this says that any sufficiently expressive system cannot prove its own consistency, rather consistency can only be proven by an outside system, which of course cannot prove its own consistency. . . .