LG-2006-02 November 25, 2006

Herbrand Logic

Timothy Hinrichs
Michael Genesereth

Stanford Logic Group
Computer Science Department
Stanford University
353 Serra Mall
Stanford, CA 94305
thinrich@cs.stanford.edu

1 Introduction

Logic was invented in an attempt to understand the foundations of mathematics. What does it mean for a theorem to be true? Does x*y really equal y*x? Paradoxes are sentences that are neither true nor false, yet they creep into mathematics; is that bad, good, or is it irrelevant? How can we prove that a mathematical proof is correct? How do we even make sense of these questions?

First-order logic (FOL) is one answer to the last question. If some theorem of a mathematical system can be proven from the axioms of that system written in first-order logic, the theorem holds. Because of this commitment to mathematical truth in general, FOL was defined so that it can be used across a broad spectrum of mathematics. It is not tailored to the study of integers, real numbers, geometry, topology, or any particular branch of mathematics.

FOL was certainly not designed to model or manipulate finite machines, the central activities in computer science. More than that, at times it is ill-suited for the purpose. Consider a detailed model of a computer which shows how its state, e.g. memory contents and program counter, changes over time. Because time in a computer is discrete, state can only change on each clock tick. A finite memory modeled for countably many time steps would require a model of countably infinite size. But in first-order logic, there is no way to constrain such a model to one of countable size; if a set of first-order sentences has a model of infinite size, it has a model of every infinite size (Lowenheim-Skolem-Tarski). Thus, in first-order logic, the query, "is the program counter positive at every time step?" is not even expressible.

Nevertheless logic is used throughout computer science, most notably in deductive database theory, constraint satisfaction, logic programming, and formal verification. Often, the foundations of these fields of study are taught as special cases of first-order logic. Here we depart from that practice (after having repeatedly taught first-order semantics only to see even the good students misunderstand the very notion of a model), and begin with a logic tailored for computer scientists, whose main interests are modeling and manipulating computers. We have employed the name Herbrand logic, to reflect its first-order syntax and Herbrand semantics.

This paper's original form was a website, http://logic.stanford.edu/∼thinrich/herbrand/html/. Part of the motivation for building the website was the frustration brought about by looking for and failing to find general results on this logic. Herbrand semantics appears repeatedly in the literature of logic programming, but the only results we were able to find in that community are for the fragment of Herbrand logic that coincides with first-order logic, i.e. $\forall*$ premises and $\exists*$ queries. After constructing the site, we realized that all the results here have been well known in the inductive theorem proving (ITP) community, e.g. [Gra05], where the central concern turns out to be proving theorems in Herbrand logic. The nonmonotonic community has also studied this logic in at least two forms, both of which fall under the umbrella of minimal entailment: Domain Circumscription and Herbrand entailment [Suc93].

Our aims are slightly different than the aims of researchers in the ITP and Nonmon communities. We are looking for ways to unify the work on deductive databases, constraint satisfaction, logic programming, and formal verification so that a system given problems stated in Herbrand logic can employ the results from these four areas to efficiently solve those problems. In this paper, we have attempted to explain some of Herbrand Logic's model theory and proof theory without slanting analysis toward one application or another. By treating it as a logic in its own right, we hope to achieve two goals: (1) provide a simple framework for building machines that can efficiently reason throughout the space of Herbrand logic and (2) improve how logic is taught to computer scientists. The intended audience of this paper is a computer science researcher who deals with logic on a regular basis; thus, it does not attempt to explain the importance of logical properties, e.g. compactness, but only attempts to formally define them and explain the differences between first-order logic and Herbrand logic.

Here is a synopsis of results on Herbrand logic.

Herbrand Logic = First-order syntax + Herbrand semantics

- Without functions, entailment is decidable.
- With functions, entailment is not semi-decidable and satisfiability is not semi-decidable.
- With functions, premises in the $\forall*$ fragment, query in the $\exists*$ fragment, entailment is semi-decidable.
- The theory of arithmetic over the natural numbers, i.e. $\langle N, <, +, * \rangle$, is finitely axiomatizable.
- Four very successful applications of logic in computer science, deductive databases, constraint satisfaction, logic programming, and formal verification, all make assumptions that are embodied in Herbrand logic: a domain closure axiom and unique names axioms.
- Inductive Theorem Proving is defined as theorem proving in Herbrand logic.

Reading Guide

- Section 2: Syntax and Semantics: Definitions for the syntax and semantics of Herbrand logic, along with syntax and semantics of first-order logic, for comparison.
- Section 3: Proof and Model Theory: Some (un)decidability results for entailment and some finite axiomatizability results.
- Section 4: Goedel: A discussion of Goedel's incompleteness proof and results showing how to finitely encode natural number arithmetic in Herbrand logic.
- Section 5: Applications: Definitions for deductive databases, constraint satisfaction problems, a couple of logic programming languages, and formal verification, all rooted in Herbrand logic.

2 Syntax and Semantics

This section defines the syntax and semantics of Herbrand logic, and for the purpose of comparison, first-order logic. It finishes with a couple of quick examples illustrating the difference between first-order semantics and Herbrand semantics.

2.1 Syntax

The syntax of Herbrand logic is exactly the same as for first-order logic.

Definition 1 (Vocabulary). *A vocabulary V consists of:*

- *A set of relation constants $\{r_1, ..., r_n\}$, each with an associated arity.*
- *A set of function constants $\{f_1, ..., f_m\}$, each with an associated arity.*

- *A non-empty set of object constants* $\{c_1, ..., c_k\}$.
- *A set of variables* $\{x_1, x_2, ...\}$.

For the sake of simplicity, we assume the set of constants in a vocabulary is finite. A countable set of constants changes the wording of certain theorems, but conceptually nothing changes.

Definition 2 (Term). *A term in V:*

- *A variable.*
- *An object constant.*
- *A function constant with arity n applied to n terms.*
- *Only expressions produced by the above rules are terms.*

Definition 3 (Sentence). *A sentence in V:*

- *A relation constant with arity n applied to n terms.*
- $(\neg\phi)$ *where* ϕ *is a sentence.*
- $(\phi \vee \psi)$*, where* ϕ *and* ψ *are sentences.*
- $(\phi \wedge \psi)$*, where* ϕ *and* ψ *are sentences.*
- $(\phi \Leftarrow \psi)$*, where* ϕ *and* ψ *are sentences.*
- $(\phi \Rightarrow \psi)$*, where* ϕ *and* ψ *are sentences.*
- $(\phi \Leftrightarrow \psi)$*, where* ϕ *and* ψ *are sentences.*
- $(\forall x.\phi)$*, where* ϕ *is a sentence.*
- $(\exists x.\phi)$*, where* ϕ *is a sentence.*
- *Only expressions produced by the above rules are sentences.*

An *atom* is a sentence of the form $p(t_1, ..., t_n)$. A *literal* is either an atom or the negation of an atom. A *ground sentence* has no variables or quantifiers. A *closed sentence* has no free variables, whereas an *open sentence* does have free variables. We treat free variables in an open sentence as being implicitly universally quantified.

2.2 Herbrand Semantics

Herbrand logic differs from first-order logic solely in the structures it considers to be models. The semantics of a given set of sentences is defined to be the set of Herbrand models that satisfy it, for a given vocabulary.

Definition 4 (Herbrand Model). *A Herbrand model for vocabulary* V *is any set of ground atoms in* V.

Definition 5 (Herbrand Satisfaction). *Let* ϕ *be a closed sentence and* M *a Herbrand model in the vocabulary* V.

- $\models_M s = t$ *if and only if* s *and* t *are syntactically identical.*
- $\models_M p(t_1, ..., t_n)$ *if and only if* $p(t_1, ..., t_n) \in M$.
- $\models_M \neg\psi$ *if and only if* $\not\models_M \psi$.
- $\models_M \phi \wedge \psi$ *if and only if* $\models_M \phi$ *and* $\models_M \psi$.
- $\models_M \phi \vee \psi$ *if and only if* $\models_M \phi$ *or* $\models_M \psi$.
- $\models_M \phi \Rightarrow \psi$ *if and only if* $\not\models_M \phi$ *or* $\models_M \psi$.
- $\models_M \phi \Leftarrow \psi$ *if and only if* $\models_M \psi \Rightarrow \phi$.
- $\models_M \phi \Leftrightarrow \psi$ *if and only if either* $\models_M \phi \wedge \psi$ *or* $\models_M \neg\phi \wedge \neg\psi$.
- $\models_M \forall x.\phi(x)$ *if and only if* $\models_M \phi(t)$ *for all ground terms* t *in* V.
- $\models_M \exists x.\phi(x)$ *if and only if* $\models_M \phi(t)$ *for some ground term* t *in* V.

An open sentence $\phi(x_1, \ldots, x_n)$ *with free variables* $x_1, \ldots, x_n$ *is satisfied by* M *if and only if*

$$\models_M \forall x_1 \ldots x_n.\phi(x_1, \ldots, x_n)$$

We always assume that satisfaction for a set of sentences Δ is defined with respect to a vocabulary that includes all the constants and variables that appear in Δ. Otherwise, a model might satisfy neither a sentence nor its negation.

One of the consequences of this definition is that the theory of equality is fixed in Herbrand logic: every ground term is distinct from every other ground term in the universe. Another consequence is that quantifiers range over exactly the set of all ground terms. That is, Herbrand logic builds in a domain closure axiom (which is sometimes infinitely long) and unique-names axioms.

Definition 6 (Herbrand Entailment). *Let* Δ *be a set of closed sentences and* V *a vocabulary that is a superset of the vocabulary of* Δ. *Let* ϕ *be a closed sentence.* Δ *entails* ϕ *with respect to vocabulary* V *if and only if every Herbrand model for* V *that satisfies* Δ *also satisfies* ϕ.

$$\Delta \models \phi \text{ wrt } V \text{ if and only if } \models_M \Delta \text{ implies } \models_M \phi, \text{ where } M \text{ is a Herbrand model for } V$$

If no vocabulary is named in satisfaction or entailment, it is assumed the minimal vocabulary is used, i.e. the vocabulary that includes just the constants in the sentences given.

2.3 First-order Semantics

To demonstrate the simplicity of Herbrand semantics, here we give the standard semantics of first-order logic for comparison.

Definition 7 (First-order Model). *A first-order model* M *consists of*

- $|M|$: *universe*
- *For each n-ary relation constant* p *an n-ary relation* p^M *over* $|M|$

- *For each n-ary function constant f an n-ary function f^M over $|M|$*
- *For each object constant c an element c^M from $|M|$*

Definition 8 (Variable Assignment). *In a model M, a variable assignment is a mapping of all the variables in the vocabulary to elements in $|M|$.*

Given an arbitrary model and a variable assignment for that model, every term in the language is assigned an element in that model's universe.

Definition 9 (e_v). *Let v be a variable assignment and M a first-order model. e_v maps a term to an element of $|M|$.*

- *For variable x, $e_v(x) = v(x)$*
- *For object constant c, $e_v(c) = c^M$*
- *For terms $t_1, \ldots, t_n, e_v(f(t_1, \ldots, t_n)) = f^M(e_v(t_1), \ldots, e_v(t_n))$*

Finally we can define satisfaction in a model. Satisfaction assumes there is some given variable assignment v.

Definition 10 (First-order Satisfaction). *Let M be a model and v a variable assignment for M. $\models_M \phi[v]$ is defined as follows.*

- $\models_M t_1 = t_2[v]$ *iff* $e_v(t_1) = e_v(t_2)$
- $\models_M p(t_1, ..., t_n)[v]$ *iff* $\langle e_v(t_1), ..., e_v(t_n)\rangle \in p^M$
- $\models_M \neg\phi[v]$ *iff* $\not\models_M \phi[v]$
- $\models_M \phi \wedge \psi[v]$ *iff* $\models_M \phi[v]$ *and* $\models_M \psi[v]$
- $\models_M \phi \vee \psi[v]$ *iff* $\models_M \phi[v]$ *or* $\models_M \psi[v]$
- $\models_M \phi \Rightarrow \psi[v]$ *iff* $\not\models_M \phi[v]$ *or* $\models_M \psi[v]$
- $\models_M \phi \Leftarrow \psi[v]$ *iff* $\models_M \psi \Rightarrow \phi[v]$
- $\models_M \phi \Leftrightarrow \psi[v]$ *iff* $\models_M \phi \wedge \psi[v]$ *or* $\models_M \neg\phi \wedge \neg\psi[v]$
- $\models_M \forall x.\phi[v]$ *iff for every d in $|M|$* $\models_M \phi[v(x/d)]$
- $\models_M \exists x.\phi[v]$ *iff for some d in $|M|$* $\models_M \phi[v(x/d)]$

Definition 11 (First-order Entailment). *Let Δ be a set of sentences and ϕ be a sentence. Δ entails ϕ if and only if every first-order model that satisfies $\Delta[v]$ for all variable assignments v also satisfies $\phi[v]$ for all variable assignments v.*

$\Delta \models \phi$ *if and only if for all variable assignments v and u,* $\models_M \Delta[v]$ *implies* $\models_M \phi[u]$, *where M is a first-order model*

2.4 Examples

The following sentence has two satisfying Herbrand models for the vocabulary $\{p, a, b\}$. It has infinitely many first-order models.

$p(a)$
Herbrand Models: $\{p(a)\}, \{p(a), p(b)\}$

Some First-order Models:
$\{\{1\}, p = \{\langle 1\rangle\}, a = 1, b = 1\}$,
$\{\{1, 2, 3, ...\}, p = \{\langle 17\rangle, \langle 63\rangle\}, a = 17, b = 51\}$,
$\{\text{Reals}, p = \{\langle 3.14159\ldots\rangle, \langle 17.0\rangle\}, a = 3.14159..., b = 0.33333...\}$

In Herbrand logic, restricting the vocabulary to $\{p, a\}$ ensures there is only one satisfying model: $\{p(a)\}$. In first-order logic, shrinking the vocabulary does not reduce the number of models at all; the only difference is that models of the smaller vocabulary do not have an assignment for b.

The following sentences are Herbrand unsatisfiable for the vocabulary $\{p, a\}$.

$$p(a)$$
$$\exists x.\neg p(x)$$

However, in first-order logic, regardless the vocabulary, they are always satisfiable, e.g. $\{\{1, 2\}, p = \{\langle 1\rangle\}, a = 1\}$. In Herbrand logic, enlarging the vocabulary to include an extra element is sufficient for satisfiability in this example: the vocabulary $\{p, a, b\}$ allows the satisfying model $\{p(a)\}$.

More generally, in first-order logic, changing the vocabulary never changes the satisfiability (again the vocabulary must always be a superset of the symbols in the axiom set), whereas in Herbrand logic, changing the vocabulary can affect satisfiability.

A note on the dependence of Herbrand satisfaction on a vocabulary. To see why satisfaction is defined as it is, suppose instead of fixing the vocabulary up front (which defines the class of candidate models), we instead define satisfaction so that the vocabulary is gleamed from the sentences we are given. Now consider the following axioms.

$$p(a)$$
$$\neg p(b)$$
$$\exists x.\neg p(x)$$

This set of sentences is satisfied by the model $\{p(a), \neg p(b)\}$ for vocabulary $\{p, a, b\}$. But, if we were to drop out the $\neg p(b)$ sentence, under the new definition of satisfaction, the vocabulary would be $\{p, a\}$, which as commented above is unsatisfiable. That is, under this new definition of satisfaction, a set of sentences could be satisfiable, but a subset of those sentences could be unsatisfiable. Clearly, that would cause havoc in proof procedures; by fixing the vocabulary up front, in this case to $\{p, a, b\}$, the satisfiability of a set of sentences ensures the satisfiability of every subset.

3 Proof and Model Theory

3.1 Proof Theory

The Proof theory for Herbrand logic is much different than the proof theory for first-order logic. The Model theory is also quite different. The proof theory is more complicated, and the model theory is much simpler.

3.1.1 Entailment

Herbrand entailment holds whenever every Herbrand model that satisfies the premises satisfies the conclusion. The natural question to ask is whether there is an algorithm that can be used to answer entailment queries, i.e. does a given set of sentences Δ entail a particular sentence ϕ? It turns out there is formally no such algorithm because Herbrand logic is expressive enough to encode Diophantine equations and their inverses, which is enough to ensure entailment is not semi-decidable. First we give the proof that shows entailment in Herbrand logic is undecidable; then we give the proof that shows entailment is not even semi-decidable. Both rely on Diophantine equations.

It is well known that solving Diophantine equations, i.e. determining whether a polynomial equation has integer roots, written $P(x_1, ..., x_n) = 0$, is undecidable. Every such equation can be expressed using multiplication and addition. For example, $4x^3y^2 + 1 = 0$ is $4 * x * x * x * y * y + 1 = 0$. The proof of the undecidability of Herbrand entailment shows how to encode addition and multiplication.

Theorem 1 ($\forall * \models \exists *$ is undecidable). *: $\Delta \models \phi$ is undecidable.*

Proof. The proof shows how to encode arithmetic: N is the set of natural numbers, represented in unary. $Add(x, y, z)$ is true when $x + y = z$. $Mult(x, y, z)$ is true when $x * y = z$.

$$\begin{array}{l}
N(0) \\
N(s(x)) \Leftarrow N(x) \\
Add(0, y, y) \\
Add(s(x), y, s(z)) \Leftarrow N(x) \wedge N(y) \wedge N(z) \wedge Add(x, y, z) \\
Mult(0, y, 0) \\
Mult(s(x), y, w) \Leftarrow N(x) \wedge N(y) \wedge N(w) \wedge Mult(x, y, z) \wedge N(z) \wedge Add(z, y, w)
\end{array}$$

Technically, we need to express addition and multiplication of all the integers, not just the natural numbers. Doing so is straightforward but tedious.

Suppose $P(x_1, ..., x_n, y)$ is the relation that says a particular polynomial with variables $x_1, \ldots, x_n$ equals y. Now we can encode the Diophantine problem.

$$\exists x_1 ... x_n . P(x_1, \ldots, x_n, 0)$$

If this sentence is entailed, certainly there is a solution to the Diophantine problem, which ensures there is a solution in the minimal Herbrand model satisfying the axioms above. Since the axioms above are Horn, all models are a superset of the minimal model, which ensures that the solution to the Diophantine is true in every model. Thus, the solution to the Diophantine is entailed by these sentences. Entailment in Herbrand logic is therefore undecidable in general. Notice also that the query is in $\exists *$ and the premises are in $\forall *$; thus, this fragment is undecidable. $\square$

Corollary 1 ($\models$ is undecidable). *: $\Delta \models \phi$ is undecidable.*

Proof. The special case defined in the last theorem is undecidable; thus, the problem in general is undecidable. □

Interestingly, we can also encode the statement that says there is no solution to the Diophantine. That problem is not semi-decidable (otherwise solving the Diophantine itself would be decidable), ensuring that Herbrand Entailment is not semi-decidable.

Theorem 2 ($\forall* \models \forall*$ is not semi-decidable). *Let Δ be a set of sentences in $\forall*$. Let ϕ be a sentence in $\forall*$. $\Delta \models \phi$ is not semi-decidable.*

Proof. Here we encode a problem that is not semi-decidable as an entailment query in Herbrand logic, with the required characteristics. More precisely, we encode the query that says a Diophantine has no solution: every candidate solution when plugged in is either less than 0 or greater than 0. Start with the axioms for less-than (lt), which are added to the axioms in the above proof.

$$\begin{array}{l} lt(s(x), 0) \\ lt(s(x), s(y)) \Leftarrow lt(x, y) \\ \neg lt(0, 0) \end{array}$$

The query that says there is no solution is then

$$\forall x_1 ... x_n y. (P(x_1, ..., x_n, y) \Rightarrow (lt(y, 0) \vee lt(0, y)))$$

That is, the query says that regardless what numbers we plug in for $x_1, \ldots, x_n$, the result is always less than 0 or greater than 0. By reasoning similar to the proof above, this sentence is entailed if and only if the Diophantine described has no solution. Because this entailment query is not semi-decidable, the premises are in $\forall*$, and the conclusion is in $\forall*$, we see that this class is not semi-decidable. □

Corollary 2 ($\models$ is not semi-decidable)). *: $\Delta \models \phi$ is not semidecidable.*

Proof. The special case defined in the last theorem is not semi-decidable; thus, the problem in general is not semi-decidable. □

So in general, there is no algorithm for determining that $\Delta \models \phi$ even if it is true. But, for a special case of Herbrand logic, entailment is semi-decidable. That fragment is semi-decidable because in it Herbrand entailment is equivalent to first-order entailment. We use $\models_{FO}$ to stand for first-order entailment.

Theorem 3 ($\forall* \models \exists*$ iff $\forall* \models_{FO} \exists*$). *: Let Δ be a set of equality-free, quantifier-free sentences and ψ be an equality-free, quantifier-free sentence.*

$$\Delta \models \exists * .\psi \textit{ if and only if } \Delta \models_{FO} \exists * .\psi$$

Proof. Below we use $\Delta \cup \neg\psi$ as shorthand for $\Delta \cup \{\neg\psi\}$.

$\Delta \models_{FO} \exists * .\psi$
$\Leftrightarrow \Delta \cup \neg\psi$ is First-Order-unsatisfiable.
$\Leftrightarrow$ there is no Herbrand model of $\Delta \cup \neg\psi$. (By Herbrand's theorem, since $\Delta \cup \neg\psi$ is quantifier-free and equality-free.)
$\Leftrightarrow \Delta \cup \neg\psi$ is Herbrand-unsat.
$\Leftrightarrow \Delta \models \exists * .\psi$

□

Corollary 3 ($\forall * \models \exists *$ is semi-decidable). *: Let Δ be a set of equality-free, quantifier-free sentences and ψ be a quantifier free sentence. $\Delta \models \exists * .\psi$ is semi-decidable.*

Proof. Since $\models$ is equivalent to $\models_{FO}$, and the latter is semi-decidable, so is the former. $\square$

Since for a problem to be semi-decidable, its complement must not even be semi-decidable, the theorem above implies $\not\models$ cannot be semi-decidable for the special case of entailment queries in the theorem. Thus, in general, $\not\models$ cannot be semi-decidable.

Corollary 4 ($\Delta \not\models \phi$ is not semi-decidable). *: $\Delta \not\models \phi$ (or equivalently, Herbrand satisfiability) is not semi-decidable.*

Proof. Because entailment and therefore unsatisfiability in the fragment with $\forall *$ premises and $\exists *$ queries is semi-decidable, satisfiability of that fragment must not be semi-decidable. Otherwise, both unsatisfiability and satisfiability would be decidable. Thus, because a special case of satisfiability or equivalently $\not\models$ is not semi-decidable, in general, satisfiability cannot be semi-decidable. $\square$

Herbrand logic in its most general form is thus incomputable: neither entailment nor its negation is semi-decidable. Consequently some true sentences require infinite proofs in Herbrand logic, making the logic inherently incomplete. (This assumes our definition of an inference system is one where checking whether a candidate proof actually proves a given query is decidable. Under this assumption, if every true sentence in a logic admits a finite proof, entailment is semi-decidable: (1) the set of all finite proofs is recursively enumerable, (2) checking whether a given proof proves the query under consideration is decidable, (3) thus, if there is a finite proof, there is always a semi-decision procedure for finding it. Since entailment is not semi-decidable, there must be true sentences without finite proofs.)

Herbrand logic is therefore more expressive than FOL. In FOL, we can encode a Diophantine, but because entailment is semi-decidable, we are ensured that there is no r.e. axiomatization of the negation of an arbitrary Diophantine since otherwise FOL would decide Diophantines, which clearly it can't. But, in Herbrand logic, we can encode both the Diophantine and its negation; obviously, the downside is that we lose semi-decidability of entailment. The benefit is that some theories that are not recursively enumerable in FOL appear to be finitely axiomatizable in Herbrand logic. See Section 4 for more information.

3.1.2 Finite Herbrand Logic

Another way to cut up the space of sentences in Herbrand logic is on the basis of what kinds of constants exist in the vocabulary. If there are no functions, the logic becomes much simpler; all the negative results above require functions in the vocabulary.

Consider a vocabulary V without function constants. Recall we assume the set of constants in a vocabulary is always finite. The set of ground terms for V is then equal to the (by definition nonempty) set of object constants, which means the universe for every Herbrand model for V is finite. The set of ground atoms is also finite, which makes the set of all Herbrand models finite. We use the term *Finite Herbrand Logic* (FHL) to refer to this class of vocabularies.

Theorem 4 (Finite Herbrand Logic Expressiveness). *: Finite Herbrand Logic has exactly the same expressiveness as Propositional Logic with a finite set of propositions.*

Proof. Let Δ be a set of FHL sentences with vocabulary V. Invent a new propositional symbol for each ground atom in V. Ground Δ. Since the set of ground terms is finite, the grounding will produce only finite-length sentences. Replace each ground atom in that grounding with the appropriate propositional symbol. The result is a set of propositional sentences that has the same consequences as Δ modulo the ground atom rewriting. The other direction is obvious: each propositional symbol is a relation constant of arity 0. $\square$

Corollary 5. *: $\Delta \models \phi$ in Finite Herbrand Logic is decidable.*

Proof. Entailment in propositional logic is decidable; hence, so is entailment in Finite Herbrand Logic. $\square$

Corollary 6. *: Every theory in Finite Herbrand Logic can be finitely axiomatized.*

Proof. It is sufficient to show every set of models can be finitely axiomatized. Every finite Herbrand model can be finitely axiomatized by conjoining all the ground literals true in the model. A vocabulary in finite Herbrand logic always has a finite number of models. Disjoining the finite axiomatizations for each of the models in the theory finitely axiomatizes that theory. $\square$

That is not to say that computationally, Finite Herbrand Logic has exactly the same properties as propositional logic; all it says is the two logics can express the same logical theories. The structure in FHL may allow for more efficient representation or processing of those theories.

3.1.3 Some More Proof Theory

In light of the negative results above, namely that Herbrand logic is inherently incomplete, it is not surprising that Herbrand logic is not compact. Recall that Compactness says that if an infinite set of sentences is unsatisfiable, there is some finite subset that is satisfiable. It guarantees finite proofs.

Theorem 5 (NonCompactness). *: Herbrand logic is not compact.*

Proof. Consider the following infinite set of sentences.

$$\begin{array}{l} p(a) \\ p(f(a)) \\ p(f(f(a))) \\ p(f(f(f(a)))) \\ \ldots \end{array}$$

Assuming the vocabulary is $\{p, a, f\}$, the ground terms are $a, f(a), f(f(a)), \ldots$, and this set of sentences entails $\forall x.p(x)$. Add in the sentence $\exists x.\neg p(x)$. Clearly, the infinite set is unsatisfiable. However, every finite subset is satisfiable with respect to the vocabulary $\{p, a, f\}$. (Every finite subset is missing either $\exists x.\neg p(x)$ or one of the sentences above. If it is the former, the set is satisfiable, and if it is the latter, the set can be satisfied by making the missing sentence false.) Thus, compactness does not hold. $\square$

Corollary 7 (Infinite Proofs). *: In Herbrand logic, Some entailed sentences admit only infinite proofs.*

Proof. The above proof demonstrates a set of sentences that entail $\forall x.p(x)$. The set of premises in any finite proof will be missing one of the above sentences; thus, those premises do not entail $\forall x.p(x)$. Thus no finite proof can exist for $\forall x.p(x)$. $\square$

This statement in this Corollary was made earlier with the condition that checking whether a candidate proof actually proves a conjecture is decidable. There is no such condition on this theorem.

Skolemization is another mainstay of many logics that does not have the usual effect in Herbrand logic.

Theorem 6. *In Herbrand logic, Skolemization does not preserve satisfiability.*

Proof. The following set of sentences are unsatisfiable.

$$\begin{array}{l} p(a) \\ \exists x.\neg p(x) \end{array}$$

Skolemizing produces a satisfiable set of sentences:

$$\begin{array}{l} p(a) \\ \neg p(k) \end{array}$$

□

This result is not surprising given the dependence of Herbrand satisfaction on the vocabulary.

3.2 Model Theory

Model theory in Herbrand logic is much simpler than it is in first-order logic.

Unlike first-order logic where two models A and B can be equivalent with varying strengths, i.e. elementarily equivalent, secondarily equivalent, ..., isomorphic, and equal, there is no such hierarchy in Herbrand Logic. Two models either satisfy all the same sentences because they are the same exact model, or they disagree on some ground atom, i.e. one satisfies it, and the other satisfies its negation. Thus, the set of ground literals satisfied by a model uniquely identify it.

In our discussion of model theory, we use the usual notation for the set of models that satisfy a given sentence set Δ: $Mod[\Delta]$. The set of sentences entailed by a set of sentences Δ is denoted $Cn[\Delta]$. Also, the set of sentences true in all models in the set M is denoted as usual by $Th[M]$.

Theorem 7. *: Let S be the set of ground literals satisfied by a Herbrand model M. S is satisfied by exactly one Herbrand model: M. Equivalently, $Mod[Th[M]] = \{M\}$.*

Corollary 8. *: Lowenheim-Skolem-Tarski does not hold in Herbrand logic.*

Proof. Lowenheim-Skolem-Tarski is a theorem in first-order logic that states if a set of sentences has a model of any infinite size, it has a model of every infinite size. Clearly this theorem does not hold in Herbrand logic, since every infinite model has countable size. □

Unlike Finite Herbrand Logic, not all theories are finitely axiomatizable.

Theorem 8. *: Some Herbrand models are not finitely axiomatizable.*

Proof. Consider a vocabulary with the set of terms $\{a, f(a), f(f(a)), f(f(f(a))), ...\}$, and a single, unary relation constant p. The set of all Herbrand models over this vocabulary is a subset of the ground atoms $\{p(a), p(f(a)), p(f(f(a)), ...\}$. There are 2^ω such subsets, which is uncountably infinite. Since there are only countably many sentences from a countable vocabulary, there are models without finite axiomatizations. □

More concretely, consider a model that encodes the digits of π, alternating between the positive literals and the negative literals.

1	4	1	5	...
$p(a)$	$\neg p(fa), \neg p(ffa), \neg p(fffa), \neg p(ffffa)$	$p(f^5a)$	$\neg p(f^6a), \neg p(f^7a), \neg p(f^8a), \neg p(f^9a)$	...

It is unlikely this model is finitely axiomatizable.

4 Goedel's Theorem

One of the benefits of Herbrand logic is that some theories that do not have finite axiomatizations, or even recursively enumerable axiomatizations, in first-order logic are finitely axiomatizable in Herbrand logic. In particular, the theory of natural arithmetic is finitely axiomatizable. Herbrand logic is therefore more expressive than FOL.

Recall that the theory of natural arithmetic is just the model $A = \langle N, <, +, * \rangle$. Its universe is the natural numbers, $<$ is an infinite table that defines the total ordering on the natural numbers; $+$ and $*$ are the usual functions that map two natural numbers onto their sum and product, respectively.

Goedel's first incompleteness proof shows that with multiplication, addition, and an ordering on integers, it is possible to encode the true sentence "this sentence is unprovable". That is, this sentence is satisfied in A. One consequence of this result is that in every complete logic, arithmetic has no recursively enumerable axiomatization.

For the purpose of contradiction, suppose there were a recursively enumerable axiomatization of arithmetic in a complete logic. Then by completeness of the logic there must be a proof of every true sentence from that axiomatization. That contradicts the truth of the sentence "this sentence is unprovable". Hence there can be no such recursively enumerable axiomatization.

In an incomplete logic that proof no longer works. Because Goedel's proof relies on representing proofs as integers, infinite proofs cannot be represented by Goedel's encoding. Goedel's statement might be more accurately translated as "this statement is not finitely provable". The proof system encoded using goedel's methodology is therefore incomplete in an incomplete logic.

To summarize, the existence of a finite axiomatization of arithmetic in an incomplete logic is consistent with Goedel's first incompleteness result. Herbrand logic appears to admit such a finite axiomatization.

What follows is an outline for how to finitely axiomatize arithmetic over the natural numbers in Herbrand logic. It is important to note that the axioms given below are the usual ones for encoding less than, addition, and multiplication; it just happens that in Herbrand logic these are sufficient to capture all of arithmetic.

Let the vocabulary be $\{0, s, <, Add, Mult\}$. First we exhibit axioms that are strong enough to capture arithmetic over the natural numbers. Then we argue that these axioms are consistent.

Theorem 9 (Axioms for $<$). *Let A be the model of arithmetic $\langle N, <, +, * \rangle$, where $N = \{0, s(0), ...\}$. Let Δ be the following set of sentences.*

1. $x < s(x)$
2. $x < s(y) \Leftarrow x < y$
3. $\neg(s(x) < x)$

4. $\neg(s(x) < y) \Leftarrow \neg(x < y)$

Then $\forall xy$. *if* $A \models (x < y)$ *then* $\Delta \models (x < y)$ *and* $\forall xy$. *if* $A \models \neg(x < y)$ *then* $\Delta \models \neg(x < y)$

Proof. Since all quantifiers range over the ground terms $0, s(0), \ldots$, it is sufficient to show the theorem for all ground terms.

First, we show that for all x, y if $A \models x < y$ then $\Delta \models x < y$. We will show for all k if $A \models k < n$ then $\Delta \models k < n$. By induction on n.

Base: $A \models k < 0$ never holds, which vacuously ensures the statement is true. $A \models k < s(0)$ only holds when $k = 0$. Axiom 1 then applies to ensure $\Delta \models k < s(0)$.

Inductive: Suppose that for all k that if $A \models k < n$ then $\Delta \models k < n$. Show for $s(n)$. Suppose $A \models k < s(n)$; two cases: $k < n$ or $k = n$. In the former, the IH ensures us that $\Delta \models k < n$. Then, we can employ axiom 2 with MP to conclude that $\Delta \models k < s(n)$. In the latter case, axiom 1 ensures that $\Delta \models k < s(n)$ since $k = n$. That finishes the proof by induction.

Second, we show that for all k $A \models \neg(n < m)$ implies $\Delta \models \neg(n < m)$. By induction on n.

Base: Suppose $A \models \neg(0 < m)$. m can only be 0. Axiom 1 ensures $\Delta \models \neg(0 < m)$.

Inductive: Assume that for all m $A \models \neg(n < m)$ implies $\Delta \models \neg(n < m)$. Show for $s(n)$. Suppose $A \models \neg(s(n) < m)$. Then either $n = m$ or $m < n$. Axiom 3 ensures $\neg(s(m) < m)$ and therefore in the first case that $\Delta \models \neg(s(n) < m)$. In the latter case, $A \models m < n$ implies $A \models \neg(n < m)$. Then, by the IH, $\Delta \models \neg(n < m)$. Axiom 4 then ensures that the IH ensures that $\Delta \models \neg(s(n) < m)$. This finishes the inductive proof. □

Theorem 10 (Axioms for +). *Let A be the model of arithmetic $\langle N, <, +, * \rangle$, where $N = \{0, s(0), \ldots\}$. Let Δ be the following set of sentences.*

1. $Add(x, y, z) \wedge Add(x, y, w) \Rightarrow z = w$

2. $Add(s(x), y, s(z)) \Leftarrow Add(x, y, z)$

3. $Add(0, x, x)$

4. $Add(x, s(y), s(z)) \Leftarrow Add(x, y, z)$

Then $\forall xyz$. *if* $A \models (x+y = z)$ *then* $\Delta \models Add(x, y, z)$ *and* $\forall xyz$. *if* $A \models \neg(x+y = z)$ *then* $\Delta \models \neg Add(x, y, z)$.

Proof. First, we show that for all xyz if $A \models x + y = z$ then $\Delta \models Add(x, y, z)$. By induction on n: for all xy. $A \models x + y = n$ implies $\Delta \models Add(x, y, n)$.

Base: $n = 0$. x and y must then be 0. Axiom 3 ensures $\Delta \models Add(0, x, x)$, which ensures $\Delta \models Add(0, 0, 0)$.

Inductive: Assume for all xy that $A \models x + y = n$ implies $\Delta \models Add(x, y, n)$. Show for $s(n)$. Suppose $x + y = s(n)$. Either x or y must be of the form $s(z)$, since otherwise their sum would not be of the form $s(n)$. Suppose it is x. Then we know $x + y = s(z) + y = s(n)$. Then clearly $z + y = n$, and by the IH $\Delta \models Add(z, y, n)$. Then using axiom 2, we are assured that $\Delta \models Add(s(z), y, s(n))$, which means $\Delta \models Add(x, y, s(n))$. The case when x is 0 and y is of the form $s(z)$ works likewise, but we use axiom 4.

Second, we show for all xyz that if $A \models \neg(x + y = z)$ then $\Delta \models \neg Add(x, y, z)$. Suppose there is some t, u, s such that $\neg(t + u = s)$ but that $\Delta \not\models \neg Add(t, u, s)$. Then there is a model M of Δ that satisfies $Add(x, y, z)$ but $\neg(x + y = z)$. Since $\models_M \Delta$, axiom 1 ensures that if $\models_M Add(x, y, w)$ then $w = z$. Now, the proof above shows that for every pair of t, u, if $A \models t + u = v$ then $\Delta \models Add(t, u, v)$. Thus, M must satisfy $Add(t,u,v)$ as well. Thus, $\models_M Add(t,u,v)$ and $\models_M Add(t,u,s)$ then $t = s$ and $t + u = s$, which is a contradiction. This completes the portion of the proof that shows Δ is strong enough to axiomatize $+$. $\square$

Theorem 11 (Axioms for $*$). *Let A be the model of arithmetic $\langle N, <, +, * \rangle$, where $N = \{0, s(0), \dots\}$. Let Δ be the following set of sentences.*

1. *Axioms of addition from above.*
2. $Mult(x,y,z) \wedge Mult(x,y,w) \Rightarrow z = w$
3. $Mult(s(x),y,w) \Leftarrow Mult(x,y,z) \wedge Add(z,y,w)$
4. $Mult(x,y,z) \Leftarrow Mult(y,x,z)$
5. $Mult(0,y,0)$

Then $\forall xyz.$ if $A \models (x{*}y = z)$ then $\Delta \models Mult(x,y,z)$ and $\forall xyz.$ if $A \models \neg(x{*}y = z)$ then $\Delta \models \neg Mult(x,y,z)$.

Proof. First, we show that for all xyz if $A \models x * y = z$ then $\Delta \models Mult(x,y,z)$. By induction on n: for all xy $A \models x * y = n$ implies $\Delta \models Mult(x,y,n)$.

Base: $n = 0$. If $x * y = 0$ then either x or y must be 0. Suppose $x = 0$. Axiom 5 states $\Delta \models Mult(0,y,0)$. If $y = 0$ but x does not, use axiom 4 to prove $\Delta \models Mult(y,0,0)$.

Inductive: Assume for $i = 1 \dots n$ that for all xy. $A \models x * y = i$ implies $\Delta \models Mult(x,y,i)$; show for $s(n)$. Suppose $x * y = s(n)$. Since $s(n) > 0$ neither x nor y can be zero, and therefore x is of the form $s(x')$. Thus, $s(x') * y = s(n)$. It is easy to see that $s(x') * y = x' * y + y$ in A. Thus $x' * y + y = s(n)$. Let $k = x' * y$. Since $y > 0$, $k < s(n)$ and the IH holds for k. That is, $\Delta \models Mult(x',y,k)$. Axiom 1 ensures that if $k + y = s(n)$ then $\Delta \models Add(k,y,s(n))$. Using $\Delta \models Mult(x',y,k)$, $\Delta \models Add(k,y,s(n))$, and axiom 3, $\Delta \models Mult(s(x'),y,s(n))$, which means $\Delta \models Mult(x,y,s(n))$. This completes the proof by induction.

Second, we show that for all xyz. if $A \models \neg(x * y = z)$ then $\Delta \models \neg Mult(x,y,z)$. By contradiction. Suppose for some t,u,v that $\neg(t * u = v)$ but that $\Delta \not\models \neg Mult(t,u,v)$. Thus, there is some model of Δ that satisfies $Mult(t,u,v)$. Since multiplication is a total function, there is an s (which is not the same as v) such that $A \models t * u = s$. Since M satisfies Δ, the above proof ensures that $\models_M Mult(t,u,s)$. Since M satisfies $Mult(t,u,s)$, $Mult(t,u,v)$, and axiom 2, s is the same term as v. Contradiction. $\square$

The above three theorems ensure the axioms given entail the theory of arithmetic over the natural numbers; it is left to show that the axioms are consistent. The difficulty with showing the axioms are consistent is that we would need to demonstrate the model that satisfies it, which is infinite and therefore cannot be demonstrated. Instead, we argue that if the rules of algebra taught in high school are consistent, so are the axioms above.

$\boldsymbol{x < s(x)}$**:** obviously $x < x + 1$ is true

$\boldsymbol{x < s(y) \Leftarrow x < y}$**:** $x < y$ (add 1 to y since x is already smaller than y) $x < y + 1$

$\boldsymbol{\neg s(x) < x}$**:** (just rewrite with $\geq$) $x + 1 \geq x$

$\boldsymbol{\neg(s(x) < y) \Leftarrow \neg(x < y)}$**:** (rewrite with $\geq$) $x \geq y$ (add 1 to x since x is already larger than y) $x + 1 \geq y$

$\boldsymbol{Add(x,y,z) \wedge Add(x,y,w) \Rightarrow z = w}$**:** functionality of addition: $x + y = z$ and $x + y = w$ means $z = w$

$\boldsymbol{Add(s(x), y, s(z)) \Leftarrow Add(x, y, z)}$**:** $x + y = z$ (add 1 to both sides) $x + y + 1 = z + 1$ (grouping) $(x + 1) + y = (z + 1)$

$\boldsymbol{Add(0, x, x)}$**:** $0 + x = x$

$\boldsymbol{Add(x, s(y), s(z)) \Leftarrow Add(x, y, z)}$**:** $x+y = z$ (add 1 to both sides) $x+y+1 = z+1$ (group) $x+(y+1) = (z + 1)$

$\boldsymbol{Mult(x, y, z) \wedge Mult(x, y, w) \Rightarrow z = w}$**:** $x * y = z$ and $x * y = w$ ensures $z = w$

$\boldsymbol{Mult(s(x), y, w) \Leftarrow Mult(x, y, z) \wedge Add(z, y, w)}$**:** $x * y = z$ and $z + y = w$ (substitute $x + y$ for z in second equation) $x * y + y = w$ (factor out the y) $(x + 1) * y = w$

$\boldsymbol{Mult(x, y, z) \Leftarrow Mult(y, x, z)}$**:** $y * x = z$ (symmetry of $*$) $x * y = z$

$\boldsymbol{Mult(0, y, 0)}$**:** $0 * y = 0$

This does not constitute a proof that the axioms are consistent, but it does give us some confidence that they are.

5 Applications

Deductive databases, logic programming, constraint satisfaction, and formal verification are often defined as special cases of first-order logic. Here we define their simplest incarnations from the perspective of Herbrand logic, where the resulting definitions are identical to those usually given in for example Ullman's *Database and Knowledge-base Systems* [Ull89] or Lloyd's *Foundations of Logic Programming* [Llo84] book. Because Herbrand semantics is simpler to teach and understand than first-order logic, we hope our introduction to the foundations of these four applications are more accessible than the first-order introduction.

5.1 Bilevel Reasoning

All four applications of Herbrand logic can be defined in a common framework: Bilevel Reasoning (BR). Bilevel reasoning focuses on reasoning about a pair $\langle \Gamma, M \rangle$, where M is a model and Γ is a set of axioms. Γ conservatively extends the model M. The idea is that the model M picks out one world in the vocabulary V and the axioms Γ extend the theory defined by M to some theory in vocabulary V', which is larger than V. It is noteworthy that in the degenerate case, V is empty, and so bilevel reasoning reduces to reasoning about a set of logical axioms.

Definition 12 (Bilevel Pair). *A bilevel pair is $\langle \Gamma, M \rangle$, where M is a Herbrand model, Γ is a set of axioms, and vocabulary$[\Gamma] \cup$ vocabulary$[M]$ is a vocabulary, i.e. every constant has exactly one arity and type, denoted vocabulary$[\langle \Gamma, M \rangle]$.*

Satisfaction for bilevel pairs can be defined using the notion of a model's reduct, which is just a way of talking about the portion of a model in a shrunken vocabulary.

Definition 13 (Reduct). *The reduct of model M wrt vocabulary V, written Reduct$[M, V]$ is the intersection of M and the ground atoms of V.*

Definition 14 (Bilevel Satisfaction). *Let $\langle \Gamma, M \rangle$ be a bilevel pair, where Γ consists of closed sentences. Let N be a model in the vocabulary $V = vocabulary[\Gamma] \cup vocabulary[M]$. N satisfies the pair exactly when N satisfies Γ wrt V and the portion of N corresponding to the vocabulary of M is the same as M.*

$$\models_N \langle \Gamma, M \rangle \text{ if and only if } \models_N \Gamma \text{ wrt } V \text{ and } Reduct[N, vocabulary[M]] = M$$

Just as the set of models for an axiomatization Δ is usually denoted $Mod[\Delta]$, we can use $Mod[\langle \Gamma, M \rangle]$ to denote the set of models that satisfy the bilevel pair $\langle \Gamma, M \rangle$. Entailment in bilevel reasoning can therefore be defined follows.

Definition 15 (Bilevel Entailment). *Let P be a bilevel pair and ϕ a closed sentence. P entails ϕ exactly when every model that satisfies P satisfies ϕ.*

$$P \models \phi \text{ if and only if for every } M \text{ in } Mod[P], \models_M \phi$$

This splitting of axioms and data is convenient, but often it is equivalent to a normal set of sentences in Herbrand logic.

Theorem 12 (Finite Bilevel Equivalent to Herbrand Logic). *: Let $\langle \Gamma, M \rangle$ be a bilevel pair, where M is finite. Then there is a sentence set Δ in Herbrand logic such that $Mod[\Delta] = Mod[\langle \Gamma, M \rangle]$.*

Proof. First we construct such a Δ. Then we show that every model that satisfies the pair satisfies it and vice versa.

Start out by including in Δ every sentence in Γ. Then for each relation r in M, perform the following. Let $\{t_1, \ldots, t_n\}$ be the set of ground tuples for which r is satisfied in M. Note this is a finite set. Include the sentence, $\forall \overline{x}.r(\overline{x}) \Leftrightarrow \overline{x} = t_1 \vee \overline{x} = t_2 \vee \cdots \vee \overline{x} = t_n$.

To show: $\models_N \langle \Gamma, M \rangle$ if and only if $\models_N \Delta$. Since Γ is included in Δ, we can remove it from both sides. Using the definition of bilevel pair satisfaction, we need now show that $M = Reduct[N, vocab[M]]$ if and only if $\models_N \forall \overline{x}.r_i(\overline{x}) \Leftrightarrow \overline{x} = t_1 \vee \overline{x} = t_2 \vee \cdots \vee \overline{x} = t_n$ for all relations r_i in $vocab[M]$.

($\Rightarrow$) Suppose $M = Reduct[N, vocab[M]]$ but for the purpose of contradiction there is some r_i such that $\not\models_N \forall \overline{x}.r_i(\overline{x}) \Leftrightarrow \overline{x} = t_1 \vee \overline{x} = t_2 \vee \cdots \vee \overline{x} = t_n$. That is, there is some $\overline{t}$ such that $\not\models_N r_i(\overline{t}) \Leftrightarrow \overline{t} = t_1 \vee \overline{t} = t_2 \vee ... \vee \overline{t} = t_n$. That is, there is either some tbar in r_i in the reduct that is not in $\{t_1, \ldots, t_n\}$ or there is some $\overline{t}$ not in r_i in the reduct but in $\{t_1, \ldots, t_n\}$. But in both cases, because the reduct of N is M, and $\{t_1, \ldots, t_n\}$ was constructed to be exactly that set of ground terms in r_i in M, neither of these can be the case. Contradiction.

($\Leftarrow$) Suppose N satisfies $\forall \overline{x}.r_i(\overline{x}) \Leftrightarrow \overline{x} = t_1 \vee \overline{x} = t_2 \vee \cdots \vee \overline{x} = t_n$ for all relations r_i in $vocab[M]$, but for the purpose of contradiction $M \neq Reduct[N, vocab[M]]$. That is there is some $r(\overline{u})$ extra in N or missing in N. Suppose the former, that $r(\overline{u})$ is in $Reduct[N, vocab[M]]$ but not M, i.e. $\overline{u}$ is not in $\{t_1, \ldots, t_n\}$. Then N could not satisfy $\forall \overline{x}.r(\overline{x}) \Leftrightarrow \overline{x} = t_1 \vee \overline{x} = t_2 \vee \cdots \vee \overline{x} = t_n$ since $\overline{u}$ is none of the t_i. Likewise, if $r(\overline{u})$ is in M but missing from $Reduct[N, vocab[M]]$ then again N cannot satisfy $\forall \overline{x}.r(\overline{x}) \Leftrightarrow \overline{x} = t_1 \vee \overline{x} = t_2 \vee \cdots \vee \overline{x} = t_n$ since this sentence entails $r(\overline{u})$. Contradiction. □

Clearly then every finite bilevel pair can be encoded as a finite set of Herbrand logic sentences. However, as we shall see, deductive databases and to a lesser extent formal verification take advantage of this separation. They along with CSPs and logic programming can be defined as subsets of a very special class of bilevel pairs–those with exactly one satisfying model.

In first-order logic, a set of axioms whose models are all elementarily equivalent, i.e. indistinguishable wrt first-order sentences, is said to be axiomatically complete. This means every sentence in the language or its negation is entailed by the axioms.

Definition 16 (Axiomatic Completeness). *Let L be a language and Δ a set of sentences. Δ is axiomatically complete if and only if for every closed sentence s in L, Δ entails s or Δ entails $\neg s$ or both.*

In Herbrand logic, if a set of sentences is axiomatically complete, those sentences are satisfied by exactly one model. (The fact that axiomatic completeness in Herbrand logic has different model-theoretic properties than first-order logic stems from the fact that Herbrand logic fixes the equality relation, whereas first-order logic does not.) Likewise, we will say that a bilevel pair is (axiomatically) complete if and only if it has a single model.

Definition 17 (Complete Bilevel Pair). *A bilevel pair P is complete if and only if $Mod[P]$ is either a singleton set or the empty set.*

Definition 18 (Complete Bilevel Axioms). *A set of bilevel axioms Γ is complete wrt vocabulary V if and only if for every model M in V such that $P = \langle \Gamma, M \rangle$ is satisfiable, P is a complete bilevel pair.*

It is sometimes natural to think of Γ as simply extending the theory defined by M, a small wrinkle on the notion of a conservative extension to a logical theory.

Definition 19 (Conservative Pair). *A pair $P = \langle \Gamma, M \rangle$ is conservative exactly when every sentence in $Cn[P]$ that is in $vocabulary[M]$ is a consequence of M itself, i.e. when it is in $Th[M]$.*

Definition 20 (Conservative Bilevel Axioms). *A set of axioms Γ is conservative wrt vocabulary V if and only if for every model M in V, $\langle \Gamma, M \rangle$ is a conservative pair.*

Deductive databases, constraint satisfaction, logic programming, and formal verification can all be defined as complete bilevel pairs.

5.2 Deductive Databases

A database (system) consists of a set of named tables and a set of logical sentences that define the contents of other tables implicitly. The tables that are stored explicitly are called the base tables and the tables that are defined implicitly by logical sentences are called the views. A database query is a logical sentence that implicitly defines a table in terms of the views and base tables in the database, i.e. it is a user-defined view. An answer to a database query is the table the view defines implicitly. In this section, our goal is to demonstrate that a database and its view definitions can be construed as a conservative, complete bilevel pair P where the (only) model that satisfies the pair is finite.

In database terminology, the names of explicitly stored tables comprise the set of extensional database predicates (EDBs), i.e. the base tables, and the remaining relations comprise the set of intensional database predicates (IDBs), i.e. the views.

In our terminology, the base tables in a database form a Herbrand model. The views are a set of logical sentences that conservatively extend that model so that the base tables plus view definitions are satisfied by a single, finite Herbrand model. Thus, we define a database as a complete, conservative bilevel pair satisfied by a single, finite model.

Since a database has a single model and therefore corresponds to a complete logical theory, the languages that have been developed for representing view definitions are tailored to take advantage of completeness. Here we discuss Datalog with negation.

Datalog has been designed to make defining a complete theory easy. Normally, defining a complete theory requires writing down a set of sentences that say which sentences are true and a set of sentences that say which sentences are false. Datalog, however, allows us to write down only those sentences that are true; its semantics ensures that all the remaining sentences are false.

This turns out to be a little tricky with negation. The rest of this section defines Datalog with negation and discusses how the *stratified* Datalog semantics ensure that every set of Datalog sentences forms a complete theory.

Definition 21 (Datalog Rule). *A Datalog rule is an implication (where traditionally* $:-$ *is used instead of* $\Leftarrow$ *and a comma is used in place of* $\wedge$*).*

$$h \Leftarrow b_1 \wedge b_2 \wedge ... \wedge b_n$$

- *The head, h, is an atomic sentence.*
- *Each element in the body, b_i, is a literal.*
- *There are no function constants.*
- *Safety: if a variable appears in the head or in a negative literal, it must appear in a positive literal in the body.*

(Usually, Datalog allows various arithmetic operations in the body of rules, e.g. addition and multiplication, but we ignore those operations here. Also, the above definition is usually called *datalog*$^{\neg}$.)

Datalog is usually viewed as a language for defining a larger database from a smaller one. It defines the contents of new relations based on the contents of the original relations, in the end producing a single model/database. Because Herbrand semantics allows multiple models to satisfy a particular set of sentences, as stated it is inappropriate for defining the semantics of Datalog rules. Three variations on Herbrand semantics are often used as the semantics for Datalog: minimal Herbrand semantics with stratified negation, stable model semantics, and well-founded semantics. We leave discussion of stable model semantics to the section on Prolog; well-founded semantics relies on 3-valued logic, which we do not address.

If a set of Datalog rules has no negation, i.e. when every literal in the body of every rule is an atom, it is Horn. It is well-known that Horn rules have a well-defined minimal model (as measured by the number of ground atoms in the Herbrand model), and the semantics for such a set of rules is defined to be that model. When the rules do include negation, the minimal model is not necessarily well-defined. For example, the sentence

$$p(a) \Leftarrow \neg q(a)$$

is logically equivalent to

$$p(a) \vee q(a)$$

which has two minimal models: $\{p(a)\}$ and $\{q(a)\}$.

While negation does present some problems for the notion of a minimal Herbrand semantics, certain restrictions on negation allow us to pick out one of the minimal models unambiguously. Stratification is one such restriction. It is defined in terms of a dependency graph.

Definition 22 (Dependency graph). *Let Δ be a set of Datalog rules. The nodes of the dependency graph for Δ are the relation constants in the vocabulary. There is an edge from r_2 to r_1 whenever there is a rule with r_1 in the head and r_2 in the body. That edge is labeled with $\neg$ whenever r_2 is in a negative literal.*

Definition 23 (Datalog Stratification). *A set of Datalog rules are stratified when its dependency graph contains no cycle through a negative edge. Stratum 0 is the set of all nodes without incoming edges. Stratum 1 is the set of all nodes adjacent to a node in stratum 0; stratum $i+1$ is the set of all nodes adjacent to a node in stratum i that are not in stratum i.*

Clearly, if there is no negation, the graph has no $\neg$ edges, which means it has a single stratum and is stratified. When a set of Datalog rules is stratified, we can always choose a single Herbrand model as its semantics. Stratum 0 contains no negation; thus it has a minimal model. The minimal model for stratum $i+1$ is constructed by extending the minimal model for stratum i by evaluating the body of all rules in stratum $i+1$ in the minimal model of stratum i.

Definition 24 (Stratified Datalog Semantics). *Let S be a set of stratified Datalog rules. Let M_0 be the minimal models that satisfies all the rules in stratum 0. To compute M_i, where $i > 0$, initialize M_i to M_{i-1}. Perform the following operations, adding the result to M_i until no changes occur.*

- *Let $h \Leftarrow b_1 \wedge ... \wedge b_n$ be a rule in stratum i.*
- *For each substitution σ such that $\models_{M_i} (b_1 \wedge \cdots \wedge b_n)\sigma$, output $h\sigma$.*

Let k be the largest stratum for any rule in S (and we're guaranteed there is such a finite k). The Stratified Datalog Semantics of S is M_k.

For example, consider the following stratified rules. (Traditionally *not* is used in place of $\neg$.)

$$\begin{array}{l} p(a) \Leftarrow \neg q(a) \\ q(b) \Leftarrow \neg t(b) \\ t(c) \end{array}$$

The minimal model for these sentences is computed as follows. Stratum 0 is just $\{t\}$; stratum 1 is $\{q\}$, and stratum 2 is $\{p\}$. For stratum 0, the minimal model is just

$$\{t(c)\}$$

The only rule in stratum 1 is the one with q in its head: $q(b) \Leftarrow \neg t(b)$. The body $\neg t(b)$ is true in the model $\{t(c)\}$. Thus, the minimal model for stratum 1 is $\{t(c), q(b)\}$ For stratum 2, the rule body of $p(a) \Leftarrow \neg q(a)$ is true in $\{t(c), q(b)\}$, making the minimal model for this set of Datalog rules $\{t(c), q(b), p(a)\}$.

Variables in negative literals can be problematic. Consider the rule set

$$\begin{array}{l} p(x) \Leftarrow \neg q(x) \\ q(a) \end{array}$$

The minimal model for stratum 0 is $\{q(a)\}$. The minimal model for stratum 1 requires computing all the x such that $\neg q(x)$ is true. If the vocabulary were known, an implementation could, when faced with such a rule, enumerate all the object constants in that vocabulary, using NAF on each grounding of the literal.

But because the vocabulary is never known in a database, no enumeration can occur. However, for the case where negative literals can always be evaluated over the known vocabulary, the minimal model can be computed.

The definition for a Datalog rule includes a statement that says for an implication to be a Datalog rule, it must obey the following constraint: if a variable appears in the head of the rule or a variable appears in a negative literal, that variable must also appear in a positive literal in the body. Now that the semantics have been defined, the reason for this constraint can be explained. Suppose we were to write a rule with a variable in the head that does not occur in the body or a rule with a variable in a negative literal that does not occur in the body.

$$p(x, y) \Leftarrow q(x)$$
$$s(a) \Leftarrow \neg r(y)$$

In the first case, suppose $q(a)$ is true. Then $p(a, x)$ is also true. If, as is sometimes the case, the set of object constants is very large or infinite (see the note at the start of the section on defining the object constants in the vocabulary of a database), the model may be very large or infinite. In the latter case, the result no longer falls under the definition of a database, as every database must be assigned a single, finite model. The constraint above removes this possibility by forcing every variable in the head of the rule to occur in a positive literal in the body. This condition is sufficient to ensure that a finite set of rules (without function constants) always has a finite model as its semantics.

In the second rule above, it is unclear how to interpret what the rule is supposed to mean. Does it mean that if there is some y for which r is false that $s(a)$ is true? Or does it mean that if $r(t)$ for every ground term t is false that $s(a)$ is true? The constraint on Datalog rules removes this ambiguity by requiring every variable in a negative literal to also occur in a positive literal in the body; by first evaluating the poositive literal, every negative literal can be ground before evaluating it using negation as failure.

Now that the syntax and semantics of Datalog have been covered, we can return to the earlier promise of defining a database in terms of a bilevel pair.

Observation 1 (Database as a Bilevel pair). *A database is a bilevel pair $\langle \Gamma, M \rangle$, where Γ is a finite set of Datalog rules that has no EDB predicate in the head of any rule.*

We now turn to some decidability results. Because Datalog does not allow function constants, the number of Herbrand models for any finite vocabulary is finite. Regardless which semantics are used, as long as the semantics assigns some subset of all Herbrand models over a finite vocabulary to a set of sentences and checking satisfaction in one of those models is decidable, entailment is decidable.

Theorem 13 (Datalog is Decidable). *Let Δ be a finite set of Datalog rules. Let S be a semantics for Datalog that chooses some subset of the Herbrand models over the vocabulary induced by Δ, where checking whether a model satisfies Δ is decidable. Let ϕ be the query.*

$$\Delta \models \phi \text{ under the semantics } S \text{ is decidable.}$$

Proof. Since Δ itself is finite, the vocabulary and therefore the number and size of all Herbrand models over that vocabulary is finite. Because satisfaction in S is decidable, to check entailment, enumerate each model for the vocabulary and when it satisfies Δ check whether it satisfies ϕ. If every model that satisfies Δ satisfies ϕ, $\Delta \models \phi$. Otherwise $\Delta \not\models \phi$. $\square$

Unlike theorem proving, answering a query in a database usually means finding all the tuples for which the query is satisfied. We call this materializing the query.

Definition 25 (Materialization). *Let Δ be a set of sentences and $\phi(x_1, ..., x_n)$ be a query with free variables $x_1, \ldots, x_n$. The materialization of the query for Δ is the set of all $\langle t_1, \ldots, t_n \rangle$ such that $\Delta \models \phi(t_1, \ldots, t_n)$, where each t_i is ground.*

Corollary 9 (Materializing Datalog is Decidable). *Same conditions as the last theorem. ϕ has the free variables $x_1, \ldots, x_n$. Materializing ϕ for Δ is decidable.*

Proof. : Let σ be some substitution $x_1/c_1, \ldots, x_n/c_n$, where each c_i is an object constant in the vocabulary of Δ. By the above theorem $\Delta \models \phi\sigma$ is decidable. If $\Delta \models \phi\sigma$, add the tuple $\langle c_1, \ldots, c_n \rangle$ into the materialization. Since ϕ is finite, the number of variables is finite; since Δ is finite, the number of object constants is finite. Thus, the number of distinct σs is finite; check entailment for each one and return the resulting set of tuples. □

5.2.1 Completeness and NAF implementation

When a theory in a complete logic, i.e. a logic with a complete proof procedure, is axiomatically complete and has a recursively enumerable axiomatization, the theory is decidable: for any query ϕ, interleave proof attempts for ϕ and $\neg\phi$. One of them is entailed, and because the logic is complete, the proof will be found. In such a theory, negation as failure (NAF) is monotonic: failing to prove ϕ ensures that $\neg\phi$ is entailed.

Definition 26 (Negation as Failure). *Negation as failure is the following (metalevel) inference rule:*

$$\frac{\Delta \not\models \phi}{\Delta \models \neg\phi}$$

NAF is monotonic whenever Δ entails either the query in question or its negation, i.e. Δ is complete for the language that includes either the query or its negation.

Theorem 14 (Monotonic NAF). *NAF is monotonic for a particular query ϕ if and only if the theory is complete wrt the language $\{\phi\}$.*

Proof.

$$\begin{aligned}
&\Delta \models \phi \vee \Delta \models \neg\phi \text{ } (\Delta \text{ is complete})\\
&\Leftrightarrow \neg(\Delta \models \phi) \Rightarrow \Delta \models \neg\phi \text{ (Logical equivalence of } p \vee q \text{ and } \neg p \Rightarrow q)\\
&\Leftrightarrow \Delta \not\models \phi \Rightarrow \Delta \models \neg\phi \text{ (Every query is either entailed or not entailed)}
\end{aligned}$$

□

Because the Stratified Datalog semantics assigns a single model to a set of rules, it ensures every theory is axiomatically complete. The above theorem ensures NAF is monotonic for every query. One benefit to these semantics is that there need be no rules with negative literals in the head, which is reflected in the definition of Datalog rules.

Since the negative portion of the theory is never explicitly written, a proof system for Datalog must rely on NAF. The definition of Stratified Semantics can be implemented directly, causing the proof system to construct the minimal model by constructing one stratum after another in a bottom-up fashion. Some implementations, on the other hand, start with the query and use the rules backwards. They evaluate negative literals in rule bodies with NAF: attempt to prove the positive version of the literal, and if that fails, conclude that the negative version must be true. If not done carefully, the proof procedure can fail to find a proof even when one exists.

For example, consider the following rule-set.

$$\begin{array}{l} p(x,y) \Leftarrow p(x,z) \wedge p(z,y) \\ q(a) \Leftarrow \neg p(a,a) \\ q(a) \end{array}$$

These rules are stratified; they have the following model: $\{q(a)\}$. The first rule says that p is transitive. The second rule says that $q(a)$ is true if $p(a,a)$ cannot be proven. The third rule says that $q(a)$ is true. If the query is $q(a)$, here is what the proof trace might look like.

$q(a)$ is true if $p(a,a)$ cannot be proven
$p(a,a)$ can be proven if $p(a,z)$ and $p(z,a)$ can be proven
$p(a,z)$ can be proven if $p(a,w)$ and $p(w,z)$ can be proven
$\ldots$

Simple iterative deepening does not help here since NAF launches a brand new proof attempt for $p(a,a)$, and iterative deepening on p(a,a) will run forever. If iterative deepening is to be used with NAF, depth limits must be shared among proof attempts.

This example illustrates a feature of the top-down approach: while answering a Datalog query is decidable, the naive top-down approach does not always decide it whereas the naive bottom-up approach will. For example, the query $p(a,a)$ may run forever. Moreover, it is hard to make the top-down approach decide such queries efficiently, even with sophisticated methods.

5.3 Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) is traditionally defined as the problem of finding an assignment of variables to values such that the assignment satisfies a given set of constraints. Here we show that a finite CSP is equivalent to a particular subclass of database queries. Infinite constraint satisfaction problems would also fall into a subclass of database queries if databases were allowed to be infinitely large; however, here we focus on finite CSPs. One significant difference between CSPs and databases is the type of answer they expect; instead of finding all solutions to the query as is done in databases, a constraint satisfaction problem is solved once any one solution is found.

Definition 27 (Constraint Satisfaction Problem). *A constraint satisfaction problem is a three-tuple* $\langle V, D_V, C_V\rangle$*:*

- V*: set of variables*
- D_V*: for each variable, a set of values, i.e. its domain*
- C_V*: a set of constraints that say which combinations of variables can be assigned which combinations of values*

A solution to a CSP is an assignment of variables to values so that every variable is assigned a value from its domain and every constraint is satisfied

Mathematically, the constraints in a CSP can be defined as tables of the allowed values, and this is the way the yearly CSP competition defines a CSP. For example if variables v_1 and v_2 have the same domain $\{a, b, c\}$ but some constraint says those variables can only be assigned different values, that constraint is mathematically equivalent to the following table.

v_1	v_2
a	b
a	c
b	a
b	c
c	a
c	b

A set of constraints is then a set of tables, where each column is labeled with a variable name. In the context of databases, these tables, each labeled C_i for some i, correspond to the EDBs. In logic, we encode the table for constraint C_i as the set of all ground atoms of the form $C_i(t_1, \ldots, t_n)$, where n is the number of columns and $t_1, \ldots, t_n$ is some tuple in the table. The CSP is solved by a variable assignment that satisfies all the constraints. The set of all such variable assignments is defined by the following conjunction. For each table C_i with columns labeled $v_1, \ldots, v_m$, include the conjunct $C_i(v_1, \ldots, v_m)$. This is a special kind of database query–a conjunctive query or equivalently a select-project-join query. Another way to think about a CSP problem is to define it as the natural join on all the constraint tables.

For example, suppose there were two of the tables as shown above: C_1 and C_2, the first labeled the same as above, v_1 on the left and v_2 on the right, and the second labeled differently, v_2 on the left and v_3 on the right. The constraint satisfaction problem would be asking for an assignment $v_1/t_1, v_2/t_2$, and v_3/t_3 so that $\langle t_1, t_2 \rangle$ is a tuple in the C_1 table and $\langle t_2, t_3 \rangle$ is a tuple in the C_2 table. The corresponding conjunctive query would be $C_1(v_1, v_2) \wedge C_2(v_2, v_3)$. All solutions to that query over those tables is the following, which is equivalent to the natural join of C_1 and C_2.

v_1	v_2	v_3
a	b	a
a	b	c
a	c	a
a	c	b
b	a	b
b	a	c
b	c	a
b	c	b
c	a	b
c	a	c
c	b	a
c	b	c

At first, this equivalence of CSPs with a subset of databases is disconcerting since solving a CSP is well-known to be NP-Complete and answering queries in databases is often considered to be polynomial. To rectify this seeming contradiction, it is sufficient to note that solving a CSP is NP-hard in terms of the number of variables. Solving a database query is polynomial in terms of the data, i.e. in terms of the size of the model; it is exponential in the number of variables in the query. Thus, database queries are in practice usually polynomial because the size of the data vastly exceeds the size of the query–the major cost in answering a database query is in manipulating the data. Not independently, the number of answers to a query is often much larger than the number of variables in the problem, which gives further support for ignoring the number of variables, as is often done in the database community. Thus, the class of queries databases usually solve are polynomial, even though theoretically the cost is exponential in the size of the query; CSPs typically consider the class of problems where the exponential cost of the query is non-negligible.

Observation 2 (CSP as a Bilevel pair). *: In the context of bilevel reasoning, a CSP is a bilevel pair $\langle \Gamma, M \rangle$, where Γ is empty, and the only query of interest is the natural join over all the relations in M (if we gave names to each argument of each relation).*

5.4 Logic Programming

This section is entitled Logic Programming, though with some apprehension. Truly this section is about Prolog and variants of it–a family of languages that were developed to program computers where all the logic is in rule form, and negation as failure is prevalent. Logic programming on the other hand is the far more general paradigm of programming a computer using formal logic itself, regardless the syntax or semantics. It is noteworthy that Prolog and its variants have become so popular that they have become synonymous to some extent with the term logic programming.

Prolog can be seen as a generalization of Datalog, and in the context of Bilevel reasoning, it is a complete bilevel pair $\langle \Gamma, M \rangle$, where M can be empty. Function constants come into the picture, certain restrictions are removed, and a more general rule form is often allowed. Declarative Logic Programming (DLP) [AB02] is one of the most general forms of logic programming.

Definition 28 (Declarative Logic Programming Rule). *A Declarative Logic Programming rule is an implication (where traditionally $:-$ is used instead of $\Leftarrow$ and a comma is used in place of $\wedge$). Because DLP mixes negation as failure and classical negation, in this definition, we use $\neg$ for classical negation and not for negation as failure.*

$$h_1 \vee ... \vee h_m \Leftarrow b_1 \wedge b_2 \wedge ... \wedge b_k, not b_{k+1} \wedge \cdots \wedge not b_n$$

- *each h_i is a classical literal, e.g. $p(a)$ or $\neg p(a)$.*
- *each b_i is a classical literal.*
- *each $not(b_j)$ is a NAF literal, e.g. $not(p(a))$ or $not(\neg p(a))$*

When compared to the definition of a Datalog Rule, (1) disjunction is allowed in the head, (2) classical negation is allowed everywhere , (3) variables are unrestricted, and (4) function constants are allowed. Prolog, the most well-known form of logic programming, does not allow (1) or (2). It is this version of logic programming we assume from this point on, making the definition as follows.

Definition 29 (Prolog Rule). *A Prolog rule is an implication.*

$$h \Leftarrow b_1 \wedge b_2 \wedge \cdots \wedge b_n$$

- *h is an atom.*
- *each b_i is a literal, e.g. $p(a)$ or $\neg p(a)$.*

Notice that just like Datalog, Prolog does not allow implications with negative literals in the head. And Just as with Datalog, a set of Prolog sentences without negation can be assigned a single, minimal model; that model happens to be the intersection of all the Herbrand models of those sentences. Negation presents difficulties in Prolog just like it did with Datalog.

The Stratified Datalog semantics works equally well with function constants, but researchers have found stratification sometimes too restrictive; two definitions for the semantics of Prolog rules with negation stand out. [GL88] introduced Stable Model Semantics, and [vGRS91] introduced Well-founded semantics. A Prolog theory under stable model semantics is not necessarily axiomatically complete, i.e. instead of assigning a single model to a set of sentences, it assigns multiple models. Well-founded semantics relies on a 3-valued logic, which in some sense compresses multiple models into a single one. Here we discuss only stable model semantics. These semantics do not naturally fit into the complete bilevel pair formalism, though the stratified semantics do.

Propositional stable model semantics relies on the notion of a stable model, which in turn relies on the Gelfond-Lifschitz transformation.

Definition 30 (Gelfond-Lifschitz Transformation). *Let Δ be a set of ground Prolog rules and M a Herbrand model in the vocabulary of Δ. The Gelfond-Lifschitz transformation of Δ with respect to M is defined as follows.*

1. *Delete any rule in Δ with a literal $\neg b_i$ in the body where $b_i \in M$.*
2. *Delete all negative literals from rules in Δ.*

Denote this by $GL[\Delta, M]$.

The result of this transformation with respect to a model M is a set of rules without negation. If M is minimal, it is a stable model.

Definition 31 (Stable Model). *Let Δ be a set of ground Prolog rules and M a Herbrand model in the vocabulary of Δ. M is a stable model of Δ if and only if M is the minimal satisfying model of $GL[\Delta, M]$.*

There will always be exactly one minimal model of $GL[\Delta, M]$ since it is Horn.

Definition 32 (Propositional Stable Model Semantics). *Let Δ be a set of ground Prolog rules. $\Delta \models \phi$ if and only if every stable model of P satisfies ϕ.*

Definition 33 (Stable Model Semantics). *Let Δ be a set of Prolog rules. Let Δ' be all ground instantiations of the rules in Δ. $\Delta \models \phi$ if and only if $\Delta' \models \phi$.*

Regardless which semantics are chosen, a query in Prolog is a single atomic sentence. If the query $\phi = p(t_1, \ldots, t_n)$ has variables, four options arise for answering the query: (1) return T if $\exists * \phi$ is entailed by the Prolog rules, (2) return some variable assignment σ such that the rules entail $\phi.\sigma$, (3) return a function that iteratively returns such variable assignments, or (4) return all such variable assignments. Part of the reason (3) and (4) are not considered in Datalog is that every Datalog query has finitely many answers, whereas in Prolog there may be infinitely many answers. Prolog implementations usually support (3) since it is the most general of the four options.

Recall that answering entailment queries in Datalog is decidable, as is finding all answers to Datalog queries. The addition of functions ensures that undecidable problems can be embedded in Prolog, even without employing negation (see the comments about encoding Diophatine equations in Section 3). That is, entailment queries for recursive Horn rules with functions is undecidable. Removing either functions or recursion regains decidability.

Theorem 15. *Entailment for Horn Prolog is undecidable.*

Theorem 16. *Entailment for non-recursive Horn Prolog is decidable.*

Theorem 17. *Entailment for function-free Horn Prolog is decidable.*

For more information on the semantics of Prolog and its variants, Lifschitz's paper, *Foundations of Logic Programming* [Lif96] surveys various semantic definitions and their computational consequences. It focuses on rules that include classical negation and on stable model semantics, closely connected to answer set programming.

5.5 Formal Verification

Formal verification at a very high level involves defining how a system works and asking whether that system obeys some property. For example, given a computer program written in C, does it ever access an element of an array before allocating the memory for that array? Here we consider a very simple class of systems: finite state machines that run forever. While this class is simple, it should be noted that it is complex enough to model the workings of any modern-day computer.

First we show how to use a bilevel pair to define a system, and then we show how to phrase queries about that system.

Definition 34 (Finite State Machine). *A finite state machine (FSM) is a tuple $\langle S, \Sigma, S_0, S_f, \delta \rangle$.*

1. *S: set of states*
2. *Σ: input vocabulary*
3. *S_0: the initial state – $S_0 \in S$*
4. *S_f: the final states – $S_f \in S$*
5. *δ: the transition function: $S \times \Sigma \rightarrow S$.*

Representing a FSM as a shown above may require a large amount of space simply because the set of states may be very large. Often, though, the states in a system need not be described as monolithyic entities but rather can be compactly described using a set of propositions, where a state is defined as the set of propositions true in the state. Those propositions will be represented by ground terms.

The initial state in the FSM can be represented as a model, M. If for example the propositions p, q, and r are true in the intial state, and only those are true in the initial state, then the model is.

$$\begin{gathered} init(p) \\ init(q) \\ init(r) \end{gathered}$$

The dynamics of most systems are the same regardless what the initial state is. Those dynamics tell us for a given state and a given input what the next state will be. To represent that proposition p is true in state s, we write

$$true(p, s)$$

To represent names for states, we will borrow from situation calculus. Each state may have many names; each name encodes the history of actions required to reach that state from the initial state. This reflects the fact that even though there are a finite set of states in a FSM, any one state may be reached arbitrarily many times by performing longer and longer sequences of actions from the initial state.

Thus, actions too take the form of propositions, and when encoded in logic the form of ground terms. If the proposition p is true in the state resulting from executing actions a, b, c from the initial state, we can express this as follows.

$$true(p, do(c, do(b, do(a, s0))))$$

Notice that the last action executed is the first one in the name of the state when read from left to right.

To tie the initial state to the state name $s0$, we need a single rule.

$$true(x, s0) \Leftarrow init(x)$$

Using simple implications, we can encode how certain actions affect the state of the world. Blocks world is a simple example. If blocks x and y are clear in state s (all of which are variables) then by executing the action $stack(x, y)$ in s, the resulting state includes the proposition $on(x, y)$.

$$\begin{gathered} true(on(x, y), do(stack(x, y), s)) \Leftarrow legal(stack(x, y), s) \\ legal(stack(x, y), s) \Leftarrow true(clear(x), s) \wedge true(clear(y), s) \end{gathered}$$

Frame axioms, sentences that say what things do not change as a result of an action, must also be encoded. For example, if block x is on block y then performing the stack action does not change that fact. (Conditioning the rule on x and y being clear ensures the action is not executed unless it is legal to do so.

$$true(on(x, y), do(stack(z, w), s)) \Leftarrow true(on(x, y), s) \wedge legal(stack(x, y), s)$$

Minimal model semantics is useful here since it ensures that every proposition that cannot be proven true in a state is in fact false in that state, without having to write down all the rules that say so.

The set of all rules that define the transition function δ also define the input vocabulary Σ. Σ is the set of all ground action names. The remaining element of a FSM that must still be defined is the set of final states, S_f. To do that, we use a distinguished unary relation *terminal*. Whenever *terminal* holds of a state, the state is a final state, i.e. a member of S_f. For example, to say the system is terminal when block a is on b and block b is on c, we could write the following.

$$terminal(s) \Leftarrow true(on(a, b), s) \wedge true(on(b, c), s)$$

To be faithful to the FSM notion of a terminal state, the rules for terminal should never distinguish states based on the state names, i.e. on the action taken to reach the current state. In other words, $terminal(s)$ should always be at the head, and no action should ever be mentioned in the body.

A system description as discussed above fits within the Bilevel reasoning framework. The initial state represents the model, and the transition and terminal rules are the axioms.

Observation 3 (System Description as a Bilevel pair). *: In the context of bilevel reasoning, a system description is a bilevel pair $\langle \Gamma, M \rangle$, where Γ defines $true(x, y)$ on top of M, the definition of the initial state, init.*

Describing a system using situation calculus allows one to define a transition function very flexibly. The example sentences given above help define a transition function as one would expect: given a state and an action, the rules say what the subsequent state looks like. But one could write more interesting rules where the state an arbitrary number of steps in the future determines what must have happened in the past.

$$true(p, s) \Leftarrow true(p, do(a, do(b, s)))$$

Such rules can easily lead to inconsistency, and are never necessary for defining a finite state machine (though sometimes such rules are convenient). Analysis of system descriptions is sometimes easier when all the logical sentences defining the transition function are Markovian: they say how an action performed in the current state affects the next state. A benefit of this restriction on system descriptions is that the consistency of the description can be enforced. GDL[LHG06] is a Markov language for describing finite, synchronous, multi-agent, discrete systems that has been used for describing hundreds of systems. It allows recursion and function constants but ensures decidability and finiteness.

The discussion so far illustrates how to construct a bilevel pair that defines the system we might like to verify. Some kinds of formal verification can be seen as just answering entailment questions about that bilevel pair.

For example, let's say we want to ask whether in every state reachable from the initial state the action a is legal.

First, we need to define which states are reachable. Again, we use Prolog to ensure we know which states are reachable and which states are not reachable.

$$\begin{array}{l} reachable(s_0) \\ reachable(do(x, s)) \Leftarrow legal(x, s) \wedge reachable(s) \end{array}$$

Then, asking whether action a is legal in every reachable state is easy.

$$\forall s.reachable(s) \Rightarrow legal(a, s)$$

Is there a legal move in each reachable state?

$$\forall s.reachable(s) \Rightarrow \exists a.legal(a, s)$$

Not every formal verification query can be phrased as a finite axiom about a bilevel pair without extending the set of ground terms (and therefore the set of models). For example, does some property p hold over every subset of reachable states? Because the set of states can be arbitrarily large (depending on the model M), any relation of fixed arity will not be able to define any subset with more elements than the arity. Even if we were to extend the language with function constants to allow lists, the set of reachable state names could be countably large, which ensures the set of all subsets is uncountable. Clearly we cannot represent uncountably large sets in Herbrand logic since every model has only countably many elements. (Of course, if we were only interested in all the finite subsets of the reachable states, that set is still countable, and we could express it in Herbrand logic.)

6 Conclusions and Future Work

Herbrand logic appears to be better suited than first-order logic for communicating to a machine problems confronted in computer science; moreover, Herbrand logic is easier to learn than first-order logic. Entailment is not semi-decidable; satisfiability is not semi-decidable; arithmetic over the natural numbers, however, can be finitely axiomatized. To some extent, deductive databases, constraint satisfaction, logic programming, and formal verification can all be formalized using small variations on Herbrand logic.

Herbrand logic was invented in an effort to understand how those four applications of logic in computer science relate in the hope that we could build machines that balance the trade-offs between the applications to more efficiently answer entailment queries. Our goal is to put some of the expertise that is required of users today for deciding whether to tackle a problem using Datalog, CSP techniques, logic programming, or formal verification techniques into the machine, thus allowing a naive user the flexibility of a non-restricted logical language while retaining the efficiency of industrially successful implementations of logic.

This quest has begun with Extensional Reasoning, where we are working on algorithms for automatically transforming a logical entailment query in finite Herbrand logic into Datalog. It is so-named because the machine must decide how to utilize a single model, i.e. a set of extensions, to efficiently answer the query at hand.

More generally, however, the notions underlying Herbrand logic can be extended in a variety of ways.

- Second-order Herbrand logic
- Finite model theory in the Herbrand framework
- Metalevel Herbrand logic
- Minimal model entailment, without reference to databases or logic programming
- Formalizing integrity constraints on databases

In the future, we plan on exploring all these avenues.

References

[AB02] Guray Alsac and Chitta Baral. Reasoning in description logics using declarative logic programming. 2002.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, 1988.

[Gra05] Bernhard Gramlich. Strategic issues, problems and challenges in inductive theorem proving. 2005.

[LHG06] N. C. Love, T. L. Hinrichs, and M. R. Genesereth. General game playing: Game description language. *Stanford Logic Group Technical Report: LG-2006-01*, 2006.

[Lif96] Vladimir Lifschitz. Foundations of logic programming. *Principles of Knowledge Representation*, pages 69–127, 1996.

[Llo84] John Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1984.

[Suc93] Marek Suchenek. First-order syntactic characterizations of minimal entailment, domain-minimal entailment, and herbrand entailment. 1993.

[Ull89] Jeffrey Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1989.

[vGRS91] Allen van Gelder, Kenneth Ross, and John Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.