

The Semantics of Metaprogramming in Prolog

DAVID SCOTT WARREN

Stony Brook University, Stony Brook, NY, USA

(e-mail: warren@cs.stonybrook.edu)

XSB Inc., Setauket, NY, USA

(e-mail: warren@xsb.com)

submitted 12 August 2024; accepted 13 September 2024

Abstract

This paper describes a semantics for pure Prolog programs with negation that provides meaning to metaprograms. Metaprograms are programs that construct and use data structures as programs. In Prolog a primary metaprogramming construct is the use of a variable as a literal in the body of a clause. The traditional Prolog 3-line metainterpreter is another example of a metaprogram. The account given here also supplies a meaning for clauses that have a variable as head, even though most Prolog systems do not support such clauses. This semantics naturally includes such programs, giving them their intuitive meaning. Ideas from Denecker and his colleagues form the basis of this approach. The key idea is to notice that if we give meanings to all propositional programs and treat Prolog rules with variables as the set of their ground instances, then we can give meanings to all programs. We must treat Prolog rules (which may be metarules) as templates for generating ground propositional rules, and not as first-order formulas, which they may not be. We use parameterized inductive definitions to give propositional models to Prolog programs, in which the propositions are expressions. Then the set of expressions of a propositional model determine a first-order Herbrand Model, providing a first-order logical semantics for all (pure) Prolog programs, including metaprograms. We give examples to show the applicability of this theory. We also demonstrate how this theory makes proofs of some important properties of metaprograms very straightforward.

KEYWORDS: inductive definitions, prolog semantics, programming language semantics, interpretation

1 Introduction

The simplest use of metaprogramming in Prolog is in the definition of the standard predicate `call/1`, which is defined with the single rule:

```
call(X) :- X.
```

This allows a Prolog programmer to construct a term at runtime into a variable, say `Goal`, then to use that data structure as a query to the Prolog engine by invoking `call(Goal)`. In most Prolog implementations, the programmer could simply use `Goal` itself instead of `call(Goal)`.

Another well-known example of a Prolog metaprogram is the 3-line metainterpreter:

```
interp(true).
interp((A,B)) :- interp(A), interp(B).
interp(H) :- rule(H,Body), interp(Body).
```

Here `rule/2` returns a “program component,” that is, a rule. This is another example of a metaprogram, in which data, the facts of the predicate `rule/2`, are treated as a program. Prolog systems normally allow programmers to retrieve the rules of the executing program by using a built-in predicate `clause/2`. By using `clause/2` the Prolog programmer has direct access to the code (rules) of the executing program.

Now consider the following “Prolog” metaprogram:

```
true.
(A,B) :- A, B.
H :- clause(H,Body), Body.
```

The program is not an ISO standard Prolog program. The first two clauses are (potentially) fine; it’s the third, with a variable as the rule head, that is problematical. But if we use our Prolog understanding, it is easy to see how this program ought to work. We can simply wrap every literal in this program with unary predicate symbol `interp/1` and we get (almost) the vanilla metainterpreter above, in which there is no restriction on the first argument of `rule/2` facts. (Note that the call to `clause/2` remains unwrapped.)

We will assume in this paper that our Prolog programs do allow variables in heads of rules. And so this is a valid Prolog program for us.

The usual semantics for Prolog is based on a logical theory: the least Herbrand Model satisfying all its Horn clauses (van Emden and Kowalski, 1976; Fitting, 1985), or a model satisfying the rules as inductive definitions of relations (Denecker et al., 2001; Denecker and Warren, 2023). These semantics do not account for metaprograms with variables in bodies or as heads of rules. This is because heads and body atoms have been treated as atomic formulas of a first-order logic, and a variable is not an atomic formula. Thus, these formulations fail for such metaprograms. And even though they can provide a meaning for our `interp/1` program above, there is no clear semantic connection between the data structures returned by `rule/2` and the program that is metainterpreted. Our semantics makes these connections and, to do so, must give more refined meanings to Prolog programs.

Rather than mapping the head and body literals to atomic formulas in first-order logic, or assertions of membership in relations, we consider them as propositional constants, requiring they be ground. If a rule has variables, it is treated as a template for all its ground instantiations. And this is where metarules, that is, those not built from first-order atomic formulas, become sets of valid propositional formulas, that is, ground rules. Note that these propositions are structured, that is, they are expressions, a.k.a., ground atomic formulas or ground terms. A propositional model of a set of propositional formulas is a set of propositions, in our case, a set of ground atomic formulas. And a set of ground atomic formulas uniquely determines a first-order Herbrand structure. In this way, we give a first-order model theoretic meaning to a Prolog (meta-)program.

To give meaning to ground programs, we use parameterized inductive definitions as proposed and developed by Denecker et al. (2001). This provides logical meanings for rules as inductive definitions and allows programs to be made up as the combination of separate components, each with an independent meaning. Thus, it provides a compositional semantics for Prolog programs. A component traditionally must contain all rules with the same head predicate symbol. In our propositional case, we have no obvious relations. We use instead any set of rules in the program and provide conditions on the set of their heads under which such rule sets can be a component.

The body of this paper first develops the semantics for positive programs, providing several equivalent characterizations of the set of propositions determined by interpreting a set of propositional rules as a parameterized inductive interpretation. It then discusses how parameterized components can be combined to construct larger components and ultimately full programs. Then the theory of parameterized components is extended to rules with negation, thereby providing a semantics for stratified programs. We omit proofs from this paper both due to space limitations and also because many of them are straightforward. The paper concludes with a discussion of the implications of this theory for metaprogramming, and for the interpretation of metaprograms in first-order logic.

2 Inductive definitions of sets of propositions

We all have encountered many inductively defined sets in our journeys through mathematics. The first example might be the set, *Nat*, of natural numbers, that is, 0, 1, 2, . . . , whose definition can be:

$$x \in Nat = \begin{cases} \text{if } x = 0 \\ \text{if } x' \in Nat \text{ and } x = succ(x') \end{cases}$$

meaning that x is a natural number if it is 0, or if it is the successor of a natural number.

Another example is the set R_a^G of nodes reachable in a graph G from node a :

$$x \in R_a^G = \begin{cases} \text{if } x = a \\ \text{if } y \in R_a^G \text{ and there is an edge in } G \text{ from } y \text{ to } x \end{cases}$$

As a simple example of a definition (perhaps not inductive but still rule-based) is the set of university requirements that a student has satisfied on their journey toward a degree. We might define (schematically for a hypothetical university):

```
met_cs_calc_reqs if took_calc_I and took_calc_II
met_cs_calc_reqs if took_calc_A and took_calc.B and took_calc.C
met_cs_math_reqs if met_cs_cals_reqs and took_discrete_math
met_cs_intro_pgming_reqs if took_pgming_I and took_pgming_II
met_cs_adv_pgming_reqs if met_cs_intro_pgming_reqs and took_algorithms
...
met_graduation_reqs if met_cs_reqs and met_distribution_reqs
```

We have left out many rules but expect the reader can envision how they could be completed. The idea is that if we plug in a student's transcript to define what

took.* propositions hold, then these rules define the set of requirements that that student has met, the goal of most students being to generate a transcript that causes `met_graduation_reqs` to show up in the set defined by these rules.

These rules can be understood as rules in a propositional logic: all the symbols in these rules (except the `if` and `and`) are propositional constants, or propositions. The set of requirements met by a particular student is an inductively defined set of propositions generated from those rules and facts. So, our goal is to precisely define propositional models for sets of rules and facts.

Thus, we turn to a general theory behind inductive definitions of sets of propositions. An inductively defined set is determined by a collection of propositional rules on a set of propositions. We let U denote the set of all propositions.

We represent a propositional rule in positive Prolog:

$$p_0 \text{ :- } p_1, p_2, \dots, p_n$$

as the slightly more abstract pair $(p_0, \{p_1, p_2, \dots, p_n\})$, turning the rule body sequence into a set. The general form of a propositional Prolog rule will be (h, B) where $h \in U$ and $B \subseteq U$. We call such a pair simply a rule. For rule (h, B) , B is the *body* of the rule and h is the *head*.

We note that the propositions in U may be structured. To handle grounded Prolog rules, we take U to be the set of all expressions (or trees) over finite string symbols. This set is the set of all legal Prolog ground terms. In logic, this set of trees is both the set of ground atomic formulas, and the set of ground terms. We will generally use the neutral word “expression” for a member of this set when we want to stress its structure; otherwise, we may call it a proposition. U will be the set of expressions throughout our development.

Definition 2.1.

(Inductive Definition) An inductive definition \mathcal{P} on set U is a set of rules on U .

We use inductive definitions on the set of propositions U to define sets of propositions. Note that since for us U is the set of expressions, an inductive definition will define a set of expressions, that is, a set of atomic formulas. And a set of atomic formulas uniquely determines a first-order Herbrand structure, that structure that is true of all and only atomic formulas in the set. In this way we can think of our inductive definitions as determining first-order (Herbrand) models. We will return to discuss this idea later.

Definition 2.2.

(A Set Closed under a Rule) For rule (h, B) , a set S is closed under (h, B) if $h \in S$ when $B \subseteq S$.

Definition 2.3.

(Set Containing A and Closed under \mathcal{P}) For inductive definition \mathcal{P} on U , and set $A \subseteq U$, a subset S of U contains A and is closed under \mathcal{P} if $A \subseteq S$ and S is closed under every rule $R \in \mathcal{P}$.

The same set of rules \mathcal{P} can determine different closed sets depending on the set A , hence we call A a *parameter set* for the rules \mathcal{P} .

Lemma 2.1.

The intersection of two sets containing A and closed under \mathcal{P} also contains A and is closed under \mathcal{P} .

Definition 2.4.

(Inductively Defined Set $F_{\mathcal{P}}(A)$) The set determined by an inductive definition \mathcal{P} on U with parameter set A , $F_{\mathcal{P}}(A)$, is the least subset of U containing A and closed under \mathcal{P} .

The previous lemma ensures that there is a unique minimal such set. Thus, $F_{\mathcal{P}}(A)$ is well-defined.

Definition 2.5.

(Head and Body Sets of \mathcal{P}) The Head Set, $H_{\mathcal{P}}$, and Body Set, $B_{\mathcal{P}}$, of rule set \mathcal{P} are:

$$H_{\mathcal{P}} = \{h : (h, B) \in \mathcal{P}\}$$

$$B_{\mathcal{P}} = \bigcup_{(h,B) \in \mathcal{P}} B$$

$H_{\mathcal{P}}$ is the set of propositions that appear in the head of any rule in \mathcal{P} . Note that the least set $F_{\mathcal{P}}(A)$ will contain only elements from $H_{\mathcal{P}}$ or A that is, $F_{\mathcal{P}}(A) \subseteq H_{\mathcal{P}} \cup A$ for any $A \subseteq U$. In Prolog we usually think of a set of clauses as defining a predicate, and all rules with that predicate in the head as determining a specific subset of that predicate. In our propositional framework, we use $H_{\mathcal{P}}$ to define the set from which the rules define a subset. And since parameter sets for rules \mathcal{P} should influence how \mathcal{P} determines a set and not add elements directly, we require throughout that parameter sets to be allowable:

Definition 2.6.

(Allowable Parameters) Given an inductive definition \mathcal{P} over U and a set $A \subseteq U$, A is an allowable parameter set for \mathcal{P} if $A \cap H_{\mathcal{P}} = \emptyset$. We require rule sets to have only allowable parameter sets.

Definition 2.7.

(Propositional Model Satisfying \mathcal{P}) A propositional model M is a set of propositions, $M \subseteq U$.

$$M \models \mathcal{P} \text{ if } F_{\mathcal{P}}((B_{\mathcal{P}} - H_{\mathcal{P}}) \cap M) = (H_{\mathcal{P}} \cap M)$$

The intuition behind this definition is the following. The model M itself provides the definition of the set A , a parameter set for \mathcal{P} . That is, $A = (B_{\mathcal{P}} - H_{\mathcal{P}}) \cap M$. Then given this parameter set, M agrees precisely with the least set determined by it and \mathcal{P} for all the head propositions of \mathcal{P} . All other propositions of U may or may not be in M .

We visualize the set of propositions of U that appear in a set of rules \mathcal{P} as shown in Figure 1. The outer box encloses all the propositions of U . The horizontal blue box contains the propositions in heads of rules, $H_{\mathcal{P}}$, and the yellow box represents the set of body propositions that are not also head propositions, $B_{\mathcal{P}} - H_{\mathcal{P}}$. The vertical salmon-colored box represents an allowed parameter set A . Note it is disjoint from $H_{\mathcal{P}}$. The darker blue sub-box of $H_{\mathcal{P}}$ and the salmon box represent $F_{\mathcal{P}}(A)$.

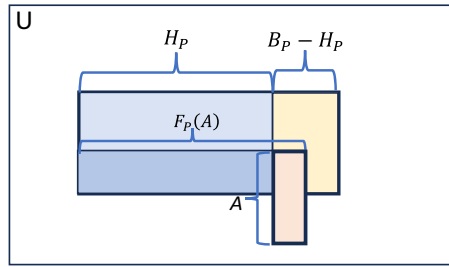


Fig. 1. Venn diagram for sets determined by rules \mathcal{P} .

We have characterized the set $F_{\mathcal{P}}(A)$ as the least set containing A and closed under \mathcal{P} . There are other ways to define this same set, and we turn to those other possible definitions.

Definition 2.8.

(The $T_{\mathcal{P},A}$ Set Function) Given an inductive definition \mathcal{P} on U and parameter set $A \subseteq U$ and set $S \subseteq U$, let $T_{\mathcal{P},A}(S) = A \cup \{h : \exists(h, B) \in \mathcal{P} \text{ such that } B \subseteq S\}$.

$T_{\mathcal{P},A}$ is a function on subsets of U that simply accumulates all the heads of rules whose body elements are all in the input set and adds all elements of A . $F_{\mathcal{P}}(A)$, the least set containing A and closed under \mathcal{P} , can be defined in terms of $T_{\mathcal{P},A}$.

Theorem 1.

$F_{\mathcal{P}}(A)$ is the least fixpoint of the $T_{\mathcal{P},A}$ operator. That is $T_{\mathcal{P},A}(F_{\mathcal{P}}(A)) = F_{\mathcal{P}}(A)$, and no proper subset of $F_{\mathcal{P}}(A)$ has this property.

We note that since $T_{\mathcal{P},A}$ is monotone operator on sets, it is known to have a least fixpoint. This theorem claims that least fixpoint is $F_{\mathcal{P}}(A)$.

We now have two definitions of $F_{\mathcal{P}}(A)$, but neither presents any obvious way to compute membership in this set. We next present two more definitions of $F_{\mathcal{P}}(A)$, which do lead to such algorithms.

Definition 2.9.

(The $T_{\mathcal{P},A^{(i)}}$ Set Operator) Let $T_{\mathcal{P},A}^{(0)} = \emptyset$; and $T_{\mathcal{P},A}^{(i+1)} = T_{\mathcal{P},A}(T_{\mathcal{P},A}^{(i)})$. Thus, $T_{\mathcal{P},A}^{(i)}$ starts with the empty set and iteratively applies $T_{\mathcal{P},A}$ to it i times.

Theorem 2.

$$F_{\mathcal{P}}(A) = \bigcup_{i=0}^{\infty} T_{\mathcal{P},A}^{(i)}.$$

This countable union indeed defines $F_{\mathcal{P}}(A)$, the least set containing A and closed under \mathcal{P} . That is if we start with the empty set and iteratively apply the $T_{\mathcal{P},A}$ operator countably many times (or until nothing new is obtained), we obtain the least set of propositions containing A and closed under the rules \mathcal{P} .

Since $T_{\mathcal{P},A}$ is monotonic, this is a non-decreasing sequence of sets. Our universal set U is infinite, so this might be an infinite sequence of ever-increasing sets and the defined set would be infinite. If $H_{\mathcal{P}}$ is finite, then $F_{\mathcal{P}}(A)$ will be finite and for some natural number n , $F_{\mathcal{P}}(A) = \bigcup_{i=0}^n T_{\mathcal{P}}^{(i)}$. A direct implementation of this construction provides a bottom-up algorithm for producing finite $F_{\mathcal{P}}(A)$.

Indeed there is yet another definition of $F_{\mathcal{P}}(A)$ that suggests another way to compute its members. The iterated $T_{\mathcal{P},A}$ algorithm computes the entire proposition set $F_{\mathcal{P}}(A)$. This next definition allows us, in some cases, to compute membership (and non-membership) in $F_{\mathcal{P}}(A)$ even when it is infinite.

Definition 2.10.

(Justification Sequence) A justification sequence for rules \mathcal{P} on U and parameter set A is a finite sequence of propositions of U , $[p_0, p_1, p_2, p_3, \dots, p_n]$ such that for every p_i in the sequence, either $p_i \in A$ or there is a rule $(p_i, B) \in \mathcal{P}$ such that $B \subseteq \{p_0, p_1, \dots, p_{i-1}\}$. That is every proposition of the sequence is either in A or is the head proposition of a rule all of whose body propositions appear earlier in the sequence.

Theorem 3.

$F_{\mathcal{P}}(A) = \{p_n : p_n \text{ is the final proposition of a justification sequence for } \mathcal{P} \text{ and } A\}$.

With this characterization of $F_{\mathcal{P}}(A)$, we can show membership of a proposition of $F_{\mathcal{P}}(A)$ by producing a justification sequence for it. A justification sequence for a given $p \in U$ can be constructed by first testing whether $p \in A$ and if not, finding a rule (p, B) and then recursively constructing a justification sequence for each $b \in B$. This produces a top-down (partial) algorithm for determining membership in $F_{\mathcal{P}}(A)$. A top-down tabling algorithm also constructs a justification sequence but guarantees in addition that no proposition appears in it more than once.

3 Parameters of inductive definitions and program components

Thus far, we have thought of parameter sets as being directly provided. But we can use a set of rules to define the parameter set for another set of rules.

Example 1.

Consider again our example of university requirements. Where requirement symbols of the form `met_*_reqs` are defined in terms of other such symbols and transcript symbols of the form `took_*`. The transcript symbols are all defined with facts, that is, rules in which the body is the empty set. For every student the requirement rules are the same, only the transcript rules differ. Thus, we can naturally keep only the transcript rules in our set \mathcal{P} rules, and let A_{Std_ID} be the set of transcript rules for the student with ID `Std.ID`. Then for any student with ID `X`, we can check whether `met_graduation_reqs` $\in F_{\mathcal{P}}(A_X)$ in order to know whether that student can graduate.

Example 2.

As another example of a parameterized inductive definition, consider transitive closure of a graph. We will represent transitive closure facts with expressions of the form `tc(X, Y)` where `X` and `Y` are integers representing nodes in a graph. And we will represent edge facts with expressions of the form `edge(X, Y)` where `X` and `Y` are integers representing nodes. So, we can represent the transitive closure of a graph with nodes with the following rules \mathcal{P} :

```
tc(X, Y) :- edge(X, Y).
tc(X, Z) :- edge(X, Z), tc(Z, Y).
```

where propositional rules are obtained by replacing X , Y , and Z in all ways consistently by integers. Thus, these rule templates generate propositional rules with propositions being expressions. We can take A to be some subset of propositions $\text{edge}(X, Y)$ (for X and Y integers). With this setup, we can determine whether (I, J) is in the transitive closure of the graph represented by the edges in A by testing whether proposition $\text{tc}(I, J)$ is in $F_{\mathcal{P}}(A)$. So, these two rule templates generate propositional rules that define the meaning of transitive closure. Then for any graph of interest represented by A , these rules will give us the correct pairs in the transitive closure of the graph.

3.1 Nested parameterized inductive definitions

Consider how we might use one parameterized inductive definition to determine a parameter set for another parameterized inductive definition. We can think of examples where this might be useful.

Example 3.

Consider again our transitive closure example:

```
tc(X, Y) :- edge(X, Y).
tc(X, Y) :- edge(X, Z), tc(Z, Y).
```

Let the corresponding set of propositional rules be \mathcal{TC} . Given a set E of expressions of the form $\text{edge}(A, B)$ for some constants A and B , we have that $F_{\mathcal{TC}}(E)$ is the transitive closure of the graph E . But perhaps our graph is over states, and the edges represent pairs of states in which the second is directly accessible from the first through some action. We want to write a definition of this set of edge pairs. So, we write a definition, that might look something like:

```
edge(X, Y) :- applicable_action(X, Op), apply_action(Op, X, Y).
```

where our edge definition checks and applies applicable actions that can move from one state to another. Let the ground rules of this program be \mathcal{MV} .

Now we want the transitive closure of this particular edge relation, the one defined by \mathcal{MV} . We use \mathcal{MV} to determine a set of edge expressions, and then use that set as the parameter set to the \mathcal{TC} program. And, assuming \mathcal{MV} doesn't need a parameter set, this is exactly the set $F_{\mathcal{TC}}(F_{\mathcal{MV}}(\emptyset))$.

This idea of using one component to generate parameter sets for another allows us to collect various program components and put them together. There are clearly some constraints on the forms of rules sets that can be combined in this way. The following theorem clarifies these constraints.

Theorem 4.

Let \mathcal{P} and \mathcal{Q} be parameterized inductive definitions over U . (We want \mathcal{Q} to generate the parameter set for \mathcal{P} .) Assume $H_{\mathcal{P}} \cap (H_{\mathcal{Q}} \cup B_{\mathcal{Q}}) = \emptyset$, that is, the head set of \mathcal{P} does not intersect with either the head set or the body set of \mathcal{Q} . Then $F_{\mathcal{P} \cup \mathcal{Q}}(A) = F_{\mathcal{P}}(F_{\mathcal{Q}}(A))$. That is the meaning of the union of the clause sets is the indicated composition, with \mathcal{Q} providing the parameter set for use by \mathcal{P} .

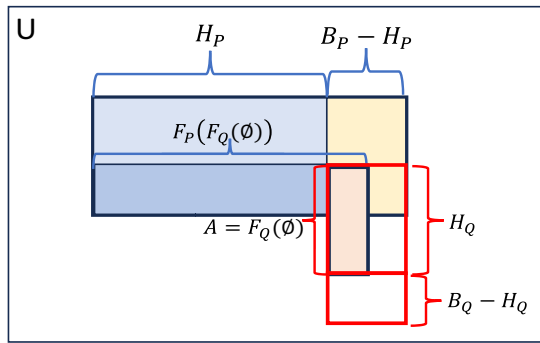


Fig. 2. Venn diagram for sets determined by parameterized \mathcal{P} .

This constraint on the use of propositions of U is clearly important. Here \mathcal{P} uses \mathcal{Q} to generate its parameter set. \mathcal{Q} cannot define or use any proposition that is potentially defined by \mathcal{P} ; it can define only propositions used by \mathcal{P} , that is, propositions in $B_{\mathcal{P}} - H_{\mathcal{P}}$. The set $(F_{\mathcal{Q}}(A))$ (of the clause set of the component below) may, and to be helpful does, intersect with the body set of the clause set of the component above. (The body sets of the two components intersecting just suggests that they both might use yet another clause set as a parameter.)

Figure 2 shows a Venn diagram for the propositions of a parameterized inductive definition \mathcal{P} with parameter set A , which is generated as the least closed set of definition \mathcal{Q} . that is, component \mathcal{Q} generates the parameter set for component \mathcal{P} . The horizontal blue and yellow rectangle represents the propositions of \mathcal{P} , as indicated. The vertical red-outlined rectangle represents the propositions of \mathcal{Q} . Its head propositions overlap with \mathcal{P} 's body (but not head) propositions. The smaller salmon-colored rectangle represents $F_{\mathcal{Q}}(\emptyset)$, as would be computed from \mathcal{Q} . This set is a subset of $H_{\mathcal{Q}}$, disjoint from $H_{\mathcal{P}}$, intersects with $B_{\mathcal{P}} - H_{\mathcal{P}}$, and constitutes the parameter set A for \mathcal{P} .

In this way we can build inductive definitions by using definitions to determine parameters to other parameterized definitions, as long as their head and body sets obey the necessary constraints. We can build large definitions by using smaller components by combining them in this way. Similarly, we may be able to decompose a large definition into subcomponents, again subject to the head and body constraints. Note that these constraints will require that a set of mutually recursive definitions must all reside in the same component. Thus, we have a compositional framework for parameterized inductive definitions.

4 Inductive definitions with negation

Prolog programs may have bodies with negative conditions, and we want to provide a semantics for such programs. However, unconstrained use of negative conditions in cycles is known to lead to complications; traditional Prolog systems go into infinite loops on programs with those cyclic uses of negation. It is known that Prolog meaningfully

evaluates the so-called stratified programs, so this is a class of programs for which we want to provide a meaning here.

We note that the (allowable) parameter set for a parameterized inductive definition does not change in the bottom-up construction of the least model of the definition. That is it can be constructed for a subcomponent independently, and then just used in the construction of the higher component. So, we can allow a test for non-membership in a parameter set assuming we can ensure that this non-membership test will not change in the least model construction. And we can ensure this by requiring that the negative goals in the rule bodies are disjoint from the head set and refer only to the parameter set. With these conditions, the non-membership conditions of negative subgoals will not change during the bottom-up construction, and the modified construction operator will converge.

We start the formalization of this idea by defining rules with negative conditions:

Definition 4.1.

(Rule with Negative Conditions) A rule on U with negative conditions is a triple (h, B, N) where $B, N \subseteq U$ and $h \in U$.

For example a Prolog rule with negative conditions, such as $p :- q, \text{ not } r, s, \text{ not } t$. would correspond to the rule $(p, \{q, s\}, \{r, t\})$.

Definition 4.2.

(The Set of Negative Condition Propositions) Given set \mathcal{P} of rules with negative conditions, the set of negative condition propositions of \mathcal{P} is:

$$N_{\mathcal{P}} = \bigcup_{(h,B,N) \in \mathcal{P}} N$$

Definition 4.3.

(Inductive Definition with Negation) An inductive definition with negation is a set \mathcal{P} of rules with negative conditions such that $H_{\mathcal{P}} \cap N_{\mathcal{P}} = \emptyset$.

With this constraint, the propositions of $N_{\mathcal{P}}$ can get their meanings only from a parameter set.

Definition 4.4.

(Closed under a Rule with Negation) A set $S \subseteq U$ is closed under (h, B, N) if $h \in S$ whenever $B \subseteq S$ and $N \cap S = \emptyset$.

Definition 4.5.

(Set Containing A and Closed under \mathcal{P} with Negations) A set S contains allowable set A and is closed under \mathcal{P} if $A \subseteq S$ and S is closed under every $R \in \mathcal{P}$.

Definition 4.6.

(Set Defined by \mathcal{P} with Negation and Set A) The set, $F_{\mathcal{P}}(A)$, determined by rule set \mathcal{P} with negation and allowable parameter set A is the least set containing A and closed under \mathcal{P} .

If $N_{\mathcal{P}} = \emptyset$, this definition reduces to the same definition we gave for positive programs. Thus, we again use $F_{\mathcal{P}}(A)$ for this set.

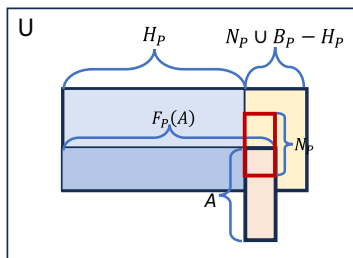


Fig. 3. Venn diagram for sets of parameterized \mathcal{P} with negation.

Figure 3 shows the structure of a component with negative rules and a parameter set. Again the large horizontal blue and yellow rectangle represents the propositions of \mathcal{P} , with the blue being the heads, and the yellow being the non-heads. $N_{\mathcal{P}}$, the negated propositions, are outlined by a red box, which is disjoint from $H_{\mathcal{P}}$. The box labeled A , a parameter set, is disjoint from $H_{\mathcal{P}}$ but intersects with $N_{\mathcal{P}}$. Propositions in that intersection are treated as true, those in $N_{\mathcal{P}}$ but not in A are considered as false. Note that those false propositions will not be in $F_{\mathcal{P}}(A)$, since they are not in $H_{\mathcal{P}}$.

We next extend the definition of the set function $T_{\mathcal{P},A}$ to include \mathcal{P} with rules with negation.

Definition 4.7.

(The $T_{\mathcal{P},A}$ Set Function) Given set of rules with negation \mathcal{P} and allowable parameter set A ,

$$T_{\mathcal{P},A}(S) = A \cup \{h : \exists(h, B, N) \in \mathcal{P} \text{ such that } B \subseteq S \wedge N \cap A = \emptyset\}$$

Definition 4.8.

(The $T_{\mathcal{P},A^{(i)}}$ Set Operator, Again) For A allowable for \mathcal{P} , let $T_{\mathcal{P},A}^{(0)} = \emptyset$; and $T_{\mathcal{P},A}^{(i+1)} = T_{\mathcal{P},A}(T_{\mathcal{P},A}^{(i)})$. Thus, $T_{\mathcal{P},A}^{(i)}$ starts with the empty set and iteratively applies $T_{\mathcal{P},A}$ to it i times.

This is same definition as for positive programs but now applies to parameterized programs with negation. And we again get the bottom-up constructive definition, this time for programs with negation:

Theorem 5.

$$F_{\mathcal{P}}(A) = \bigcup_{i=0}^{\infty} T_{\mathcal{P},A}^{(i)}$$

A valid decomposition of a program with negation provides a stratification of the propositions of the program. Propositions that are parameters to a component are in lower strata than propositions in the heads of rules in the component. The framework here supports a somewhat stronger form of predicate stratification, but not full local stratification. Predicate stratification requires that the rules can be partitioned using the root symbol of each expression in the head of any rule. (This implies that a program with a variable as a root head is not predicate stratifiable and not stratifiable by our definition either.) Our definition requires that heads of parameters not be unifiable with heads of the containing rules.

Recall that Prolog adds the builtin rules:

`true.`

`(A,B) :- A, B.`

to describe the semantics of rule bodies. Prolog programs are expressions and so rules are expressions and so to support metainterpretation, the rules bodies must be given meanings, and that is what these two rules do. Now we have added a new form for rule bodies, those containing negative literals. So for example the rule $(p, \{q, t\}, \{r, s\})$ (written in the form in our definitions with a positive set and a negative set) can be written in Prolog as `p :- q, not(r), not(s), t.` (Other orders of the body literals would also be a Prolog form of this rule.) So negative body literals are represented by trees with root `not/1`. Thus, we need another rule in the metainterpreter:

`not(X) :- not(X).`

This looks like a tautology but is not; the `not` in the head is the root of an expression; the `not` on the right must be treated by the Prolog evaluator as indicating a negative body literal. that is, this syntax must be interpreted by the Prolog evaluator as the rule $(\text{not}(X), \{\}, \{X\})$. This allows Prolog to metainterpret programs with negative body literals.

Unfortunately, our definition of stratified negation is not strong enough to support this metarule. Any program containing this rule, indeed this rule itself, is not stratifiable. Note that `not(a)` is an instance of the head of this rule, and with X instantiated to `not(a)` it is also in the set of negated bodies of instances of this rule. So there is a nonempty intersection and the instance of this rule cannot be stratified. Thus, a complete semantics for metaprogramming with Prolog programs with negation must await a treatment of nonstratified negation. The work Denecker and Vennekens (2014) on a well-founded semantics for inductive definitions with negation should be directly applicable here and would solve this problem.

5 First-order models of (meta-)programs

Thus far, we have defined the least propositional model for a set of propositional rules. Recall that the propositions are, in fact, expressions, that is, ground atomic formulas in a first-order logic over an appropriate language. As such, the set of ground atomic formulas uniquely determines a first-order Herbrand model. We can take that first-order model to characterize the meaning of the given Prolog program. This is formalized in the definition:

Definition 5.1.

(First-Order Herbrand Model of \mathcal{P}) Let M be a first-order Herbrand structure over a super-language of \mathcal{P} . Let M be the set of ground atomic formulas true in M . Then

$$M \models \mathcal{P} \text{ if } M \cap H_{\mathcal{P}} = F_{\mathcal{P}}((B_{\mathcal{P}} - H_{\mathcal{P}}) \cap M)$$

That is, if we use the atomic formulas true in M to determine an allowable parameter set, A , then on the heads of \mathcal{P} , M is true for exactly the atomic formulas in the least

set containing A and closed under \mathcal{P} . It is the case that models of the composition of components is the intersection of the models of the components.

Consider an example of a metarule and its meaning in a first-order model:

```
truly_believes(X,P) :- believes(X,P), P.
```

This rule says that if someone believes a proposition P and P is true, then that someone has a true belief of P . Note that the first two occurrences of P in this rule are first-order terms, while the final occurrence is an atomic formula.

Now consider a ground instance of this rule:

```
truly_believes(david,tall(marc)) :- believes(david,tall(marc)),
tall(marc).
```

Considered as a first-order statement, the term `tall(marc)` is being used as a name (an object in the domain of the structure) for the atomic formula `tall(marc)` in the logic. This is a kind of “Goedel numbering” the *structure* `tall(marc)` serves as a name for the formula `tall(marc)`. Since the elements of our Herbrand domain, that is, expressions, look so much like atomic formulas in our logic, Prolog uses this “identity” function as its Goedel formula-naming function.

Every term in a Prolog program is the Goedel expression of a ground atomic formula. Prolog also treats `not(X)` as the name for the negation of the ground literal `not X`. (This means, of course, that `not` cannot be a predicate symbol.)

6 Discussion

The semantics presented in this paper makes proofs of some program equivalences straightforward. Consider the transform that takes a Prolog program and wraps every atomic formula in every rule with a new unary predicate, say `holds/1`. Prolog programmers know Prolog will give the same answers (modulo the top `hold` wrapper) for the same queries. Note this works for programs with negation with our semantics, whereas it would not under predicate stratification. It is easy to see from our semantics that expression `holds(E)` is in the meaning of the wrapped program if and only if E is in the meaning of the first.

As another example, Prolog programmers know that Prolog will give the same result for asking a query X as it does when asking `call(X)`. Why this is true is clear from the definition of `call/1`, that is, `call(X) :- X`.

Recall our early metainterpreter example:

```
true.
(G1,G2) :- G1, G2.
G :- clause(G,Body), Body.
```

The well-known 3-line Prolog metainterpreter is obtained from this program by wrapping every subgoal (variable or otherwise) with new predicate `interp/1`. The reasoning for `holds/1` applies here. The `clause(G,Body)` subgoal is not wrapped for computational reasons, not semantic ones.¹

¹ Try it wrapped. You will find it leads Prolog into an infinite loop.

We note that a program with a rule with a variable head cannot have a nonempty parameter set and thus cannot be decomposed. A parameter set must be disjoint from the head set, yet the head set of such a program is all of U . The fact that such programs cannot be understood as made up of components may be a good reason for Prolog implementations to disallow them. And wrapping such rule (sub)sets with a new unary symbol allows the Prolog programmer to effectively get around this limitation when desired.

7 Why yet another semantics for Prolog?

Do we need yet another semantics for Prolog? The traditional minimal Herbrand model semantics for Prolog is based on first-order logic (FOL) and its models and proof theory. This is appropriate for Prolog as a language for representing data and knowledge since FOL is the accepted foundation for knowledge representation (KR) languages. But Prolog is also a programming language for implementing algorithms and was clearly understood as such in its early days (Warren et al., 1977; Kowalski, 1979a, 1979b). The semantics of this paper is based on simple set induction, a basic computational paradigm. Induction is fundamentally about computing; FOL is fundamentally about static knowledge representation.

This induction semantics is lower-level, more primitive, more abstract, than the logic-based semantics. This induction semantics can be used to define the minimal model semantics, as described in Section 5. And it is clear why: a complete proof system is an inductive definition of the set of logically true expressions, exactly an inductive definition. But to allow FOL to give an account of Prolog, the non-first-order concept of minimality, or closed world assumption, must be added. This add-on that is necessary for FOL is intrinsic to inductive definitions.

Also note that another important application of Prolog after KR is grammars. And grammars are introduced to students in courses on the theory of computation, not as statements in a logic, but as inductive definitions of sets of strings. Undoubtedly there is a close relationship between logic and grammars (e.g., Pereira and Warren (1983)), but it seems clear that grammars are fundamentally first understood as inductive definitions.

Teaching the semantics of Prolog to students without a strong background in logic would be much easier with this inductive semantics. The student need not be taught FO formulas, quantification, FO models, truth in a model, satisfiability, proofs, general substitutions, Skolemization, disjunctive normal form, resolution, Horn clauses, refutation proofs, and finally Herbrand models (in which the previously intuitive FO concept of a function gets specialized into a very non-intuitive particular function). These are all very important concepts, and they should be understood by students of knowledge representation. But, they are not required by students who simply want to understand the meaning of a Prolog program as they would understand the meaning of a Lisp program.

And finally, this semantics is compositional, as any self-respecting programming language semantics simply must be. Programmers understand their programs by understanding their programs' pieces. A semantics simply must account for this.

Denecker et al. (2001) and Warren and Denecker (2023) emphasized the importance of compositionality and we have followed his lead.

8 Related work

There has been much work on metaprogramming in logic programming. We discuss briefly work deemed relevant to ours.

Costatini (2002) provides a good survey of a number of early approaches to metaprogramming in logic. Bry (2020) explores metalogic in the context of a much larger fragment of logic than just Prolog.

The idea of basing Prolog semantics on a least fixpoint dates back to van Emden and Kowalski's seminal paper (van Emden and Kowalski, 1976), in which the $T|_{\mathcal{P}}$ operator was defined. It was further elaborated by Lloyd (1984).

Our approach to the semantics of Prolog most closely follows that of Denecker et al. (2001) and Denecker and Warren (2023) in emphasizing parameterized inductive definitions. Our approach here agrees with Denecker's on Prolog programs that are in first-order logic. But we also handle metarules (non-first-order rules), treating them as templates for propositional programs. Also, our treatment of negation is more restrictive than Denecker's, as noted, but we feel ours may have clearer motivations being based on parameterized programs.

9 Conclusion

In this paper we have presented a compositional semantics for pure, (finitely locally) stratified Prolog programs and metaprograms. (We have mentioned how the stratification might be strengthened.) The semantics given here is a simple generalization of Denecker's semantics of Prolog as inductive definitions (restricted to stratified programs). By treating rules with variables as templates for propositional rules, we are able to define a first-order model for Prolog meta-programs. Their meanings agree exactly with what Prolog interpreters do and with Prolog programmers' intuitions.

We want to emphasize the importance of compositionality of Prolog programs, that they can be constructed from components with independent meanings. Serious programming languages need to be compositional, that is programs are made up of independently understandable components. With a truly non-compositional language, the programmer must keep an entire program in mind when trying to develop a solution to a problem. And for large programs, that is not possible. Experientially, Prolog programmers can and do write very large programs, one piece at a time. Prolog must be compositional. But the traditional least model semantics is not compositional; one needs the entire program to define its least model. This closed world assumption must be applied to the entire program. It is the world that is being closed.

But in this inductive theory, each component has its own closure assumption, the one that comes with the idea of induction. Each component has its own "closed-neighborhood" assumption. And each component has its own meaning. Basing the theory on parameterized inductive definitions is what makes this possible.

Acknowledgements

I would like especially to thank Marc Denecker for his tireless commitment to interacting with me (and educating me) on important aspects of formal logic and its uses in knowledge representation and programming.

References

- BRY, F. 2020. In praise of impredicativity: A contribution to the formalization of meta-programming. *Theory and Practice of Logic Programming* 20, 1, 99–146.
- COSTANTINI, S. 2002. Meta-reasoning: A survey. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II. Lecture Notes in Computer Science*, KAKAS A. C. and SADRI F., Eds. vol. 2408, Springer, 253–288.
- BRUYNNOOGHE, M., DENECKER, M. AND MAREK, V. W. 2001. Logic programming revisited: Logic programs as inductive definitions. *ACM Transactions on Computational Logic* 2, 4, 623–654.
- DENECKER, M. AND VENNEKENS, J. 2014. The well-founded semantics is the principle of inductive definition, revisited. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, KR 20-24 July 2014*, C. Baral, G. D. Giacomo, and T. Eiter, Eds. AAAI Press, Vienna, Austria, 22–31.
- DENECKER, M. AND WARREN, D. S. 2023. The logic of logic programming. *CoRR abs/2304.13430*. <https://doi.org/10.48550/arXiv.2304.13430>
- FITTING, M. 1985. A deterministic prolog fixpoint semantics. *The Journal of Logic Programming* 2, 2, 111–118.
- KOWALSKI, R. A. 1979a. Algorithm = logic + control. *Communications of the ACM* 22, 7, 424–436.
- KOWALSKI, R. A. 1979b. Logic for problem solving. In *The Computer Science Library: Artificial Intelligence Series*, vol. 7, North-Holland.
- LLOYD, J. W. 1984. *Foundations of Logic Programming*. 1st Edition, Springer.
- PEREIRA, F. C. N. AND WARREN, D. H. D. 1983. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics, Massachusetts Institute of Technology, Cambridge, 15-17 June. 1983*, Massachusetts Institute of Technology, Cambridge, MA, USA, 137–144.
- VAN EMDEN, M. H. AND KOWALSKI, R. A. 1976. The semantics of predicate logic as a programming language. *Journal of the ACM* 23, 4, 733–742.
- WARREN, D. H. D., PEREIRA, L. M. AND PEREIRA, F. 1977. Prolog - the language and its implementation compared with lisp. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages, 15-17 Aug. 1977*, J. Low, Ed. ACM, USA, 109–115.
- WARREN, D. S. AND DENECKER, M. 2023. A better logical semantics for prolog. In *Prolog: The Next 50 Years. Lecture Notes in Computer Science*, D. S. Warren, V. Dahl, T. Eiter, M. V. Hermenegildo, R. A. Kowalski, and F. Rossi, vol. 13900, Springer, 82–92.