Proof-theoretic Foundations of Normal Logic Programs

ELMER E SALAZAR, University of Texas at Dallas
GOPAL GUPTA, University of Texas at Dallas

There are several semantics based in logic programming for negation as failure. These semantics can be realized with a combination of induction and coinduction, and this realization can be used to develop a goal-directed method of computing models. In essence, the difference between these semantics is how they resolve the unstratified portions of a program. In this paper, while restricting ourselves to the propositional case, we show how a semantics is a mixture of induction and coinduction, and how we can use coinduction to resolve the cycles formed by the rules in a program. We present denotational semantics based on a fixed point of a function, and show its equivalence to the use of induction and coinduction. We take a look at the different ways a semantics may resolve cycles, and show how to implement two popular semantics, well-founded and stable models, as well as costable model semantics. Finally, we present operational semantics as a parametrized goal-directed algorithm that allows us to determine how cycles are resolved.

CCS Concepts: •**Theory of computation → Constraint and logic programming;**

Additional Key Words and Phrases: negation-as-failure, proof-theoretic, non-monotonic semantics,etc.

ACM Reference format:
Elmer E Salazar and Gopal Gupta. 2016. Proof-theoretic Foundations of Normal Logic Programs. 1, 1, Article 1 (January 2016), 53 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Considerable amount of research has been done on adding negation to logic programming over the last 40 years [1, 17]. Many semantics have been proposed: well-founded semantics, Fittings 3-valued semantics, the stable model semantics, perfect model semantics, etc. Dix [3, 4] has done a systematic study of these semantics, proposing a number of properties that can be used to characterize a semantics. In this paper we show that various semantics of negation can be more elegantly characterized via a combination of induction and coinduction. Induction captures well-founded computations while coinduction captures cyclical, consistent computations. Various semantics are a combination of the two. They differ in what value they assign to cyclically dependent computations. For example, given a cycle of calls where p calls q and q calls p, then the well-founded semantics assigns p and q the value false, the Fitting 3-valued semantics assigns $\perp$ (unknown), and the stable model semantics false.

Induction and coinduction both have an operational semantics, based on recursion and co-recursion, respectively. Thus, our characterization of these semantics based on induction and coinduction also results in elegant, query-driven execution strategies discussed later. The ultimate benefit of this insight is that practical goal-directed execution strategies have been designed for predicate answer set programming [9, 15].

DOI: 10.1145/nnnnnnn.nnnnnnn

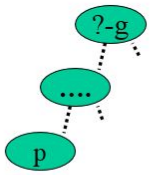
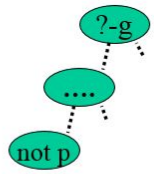
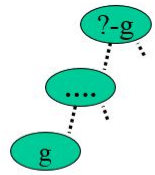
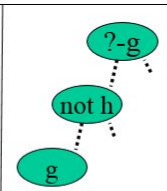
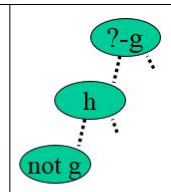
In this paper we give the declarative and operational semantics for various semantics of normal logic programs in a unifying, systematic manner. We consider four semantics for normal logic programs: Fitting's 3-valued semantics, well-founded semantics, stable model semantics, and co-stable model semantics. Our systematic, unifying characterization not only increases our understanding of various semantics of normal logic programs, it also allows us to produce efficient, query-driven implementation of these semantics.

The intuition for our work, loosely speaking, is the following. During execution of a query wrt a logic program, the execution can be well-founded or it can contain cycles that can keep unfolding forever. If the execution is well-founded then all the goals will get resolved during a successful top-down execution of the query g, with the final goal in the final resolvent matching a fact. This case will result in successful execution of the goal g. Alternatively, the terminal call will be of the form `not p` with no matching rules for p. In such a case, `not p` will succeed and query will be resolved successfully. Essentially, if the execution is well-founded, i.e., there are no infinitely unfolding cycles, then there is a single, unique model for the program [1]. All semantics of negation will find this single, unique model. If the execution of g is not well-founded, then loops (over negation) will arise. In such a case, different semantics of negation (well-founded semantics [20], stable model semantics [6], Fitting's 3-valued semantics [5], co-stable model semantics [7], etc.) will make different choices in different situations. If we have goal g, and during execution, a recursive call to g is encountered again resulting in a potentially infinitely unfolding computation, then there can be multiple possibilities (in all cases, we assume that the program is completed and that only supported models are considered):

(1) There are no intervening negative calls between the query g and the recursive call g: Multiple possibilities exist in such a case and so multiple values for g are possible: $\bot$ (Fitting's 3-valued semantics), False (well-founded semantics, stable model semantics, and co-stable model semantics), or True (co-stable model semantics).
(2) There are even number of intervening negations between g and its recursive call: In such a case, multiple models are possible. Indeed, the well-founded semantics and Fitting's 3-valued semantics will assign $\bot$, while the stable and co-stable model semantics will assign true to g in one world and will assign false in another.
(3) Query g leads to a recursive call to g with odd number of intervening negations: in such a case, the values possible for g are $\bot$ (Fitting's 3-valued and well-founded semantics) or False (stable model and co-stable model semantics). In the latter case, the conjunction of goals leading from the query g to recursive call should be false. If this conjunction evaluates to true, then a model cannot exist.

The above intuition is summarized in figure 1.

The rest of the paper is organized as follows: In Section 2 we provide a review of negation-as-failure, CoSLD resolution , and various semantic definitions that are of interest for this paper. We will also formally define the language we will be working with. In section 3 we describe a declarative coinductive representation of semantics and prove that all considered semantics can be represented in this way. Section 4 presents the declarative semantics, and in section 5 we give the operational semantics in the form of a query-driven algorithm. Finally, Section 6 briefly describes Dix's work and founded semantics[12], and how they are related to our work.

				
p is a fact: Well-founded comp.	p is not defined: Well-founded comp.	positive loop: no intervening not.	even loop: intervening not.	odd loop
No choice: Same model for all semantics	No choice: Same model for all semantics	Many possibilities Assign False Assign ⊥ Assign True	Many possibilities Assign ⊥ Assign True Assign False	Assign ⊥ *or, only way a model exists if h is false and g is false*
p: true g: true	p: false g: true	F: WFS, SM, coSM ⊥ : Fitting True: coSM	⊥ : Fitting, WFS True: SM, coSM False: SM, coSM	⊥ : Fitting, WFS SM & coSM will falsify g & h

WFS = Well Founded Semantics; SM = Stable Model Semantics

Fig. 1. Commonalities Among Semantics

2 BACKGROUND

2.1 Negation-as-Failure and the Language

Negation-as-failure is an interpretation of negation stemming from the closed world assumption and adds a new global axiom: if a proposition is unable to be proved assume it is false. The completion of a program is a way of identifying supported models and handling negation-as-failure[2].

Throughout this paper we will represent the negation of a proposition p as **not** p to indicate we are working with negation-as-failure.

Definition 2.1. A *literal* is a proposition or its negation. For some literal L, prop(L) is the proposition the literal is constructed from.

If prop(L) = L then we say L is positive. Otherwise, L is negative and **not** L = prop(L).

Definition 2.2. A *program* is a set of *rules* R of the following form:

$$H{:-}B_1, B_2, \cdots, B_n, \textbf{not } B_{n+1}, \textbf{not } B_{n+2}, \cdots, \textbf{not } B_{n+m}.$$

where $n, m \geq 0$, and $H, B_1, B_2, \cdots, B_{n+m}$ are propositions.

In addition, for convenience we define the following functions:

- head(R) = H,
- pos(R) = $\{B_1, B_2, \cdots, B_n\}$,
- neg(R) = $\{B_{n+1}, B_{n+2}, \cdots, B_{n+m}\}$,
- props(R) = $\{H\} \cup$ pos(R) $\cup$ neg(R),
- body(R) = pos(R) $\cup$ {**not** p | p $\in$ neg(R)}
- for some program P, props(P) = {p | $R \in P$, p $\in$ props(R)}, and
- for some program P, lit (P) = props(P) $\cup$ {**not** p | p $\in$ props(P)}.

A *fact* is a rule (written as p.) for which no B_i exists. That is pos(R) $\cup$ neg(R) = {}.

```
p :- s.
p :- not q.
q :- not p.
r :- p.
```

Program 1. Even Cycle

We will be focusing on semantics that agree with the completion. In horne logic, a rule is interpreted as an implication where the body implies the head. The completion of a program interprets a set of rules with the same head as a bi-implication with the head on one side and the disjunction of the bodies on the other. This agrees with the axiom: if a proposition cannot be proved assume it is false.

Definition 2.3. Let P be a program. We can represent all facts as having a body of true and any proposition that is not the head of some rule we can imagine a rule with a body of false. Then, for all propositions $\mathrm{p} \in \mathrm{props}(P)$, let B be a disjunction of conjunctions such that each conjunction in B is the body of some rule in P with p as the head, and B contains all such conjunctions. Then, $\mathrm{p} \iff B$ is the *completion rule* for p. The *completion* of P is the set of all such completion rules.
In addition, we will assume $\bot \iff \bot$ is true.

Definition 2.4. Let $\mathcal{S}$ be some semantics. Then $\mathcal{S}$ is said to be a *completion semantics* if and only if for all programs P, every model with respect to $\mathcal{S}$ is also a model of the completion of P.

The completion of a program can be simulated by adding new rules called dual rules to the program. For each proposition p in a program we can add a new symbol **not** p and rules for **not** p so that **not** p is true if and only we cannot prove p. The resulting program is called the extended program.

Definition 2.5. For some program P, the *extended program*, $\mathrm{ext}(P)$, is defined by extending P as follows:
For each proposition $\mathrm{p} \in \mathrm{props}(P)$:

- If p is not the head of any rule in P, then add a fact for **not** p.
- If there is a fact for p in P, then ignore p.
- Otherwise, take the body of the Clark's Completion rule for p, negate it, and use De Morgan's Law and distribution until it is a disjunction of conjunctions. For each conjunction, add a rule with **not** p as the head and the conjunction as it's body.

As an example consider program 1. The extended program is generated by adding the rules:

```
not p :- not s, q.
not q :- p.
not r :- not p.
not s.
```

As can be seen, the only difference between how a program and an extended program are defined is the fact that extended programs have negated literals in the head. We will extend our representation for programs to account for that.

Definition 2.6. Let P be a program. For each rule $r \in \mathrm{ext}(P)$ with a negative literal in the head, r is of the form:

$$\textbf{not}\ H{:-}B_1, B_2, \cdots, B_n, \textbf{not}\ B_{n+1}, \textbf{not}\ B_{n+2}, \cdots, \textbf{not}\ B_{n+m}$$

where $n, m \geq 0$, and $H, B_1, B_2, \cdots, B_{n+m}$ are propositions. In addition,

- head(r) = **not** H,
- pos(r) = $\{B_1, B_2, \cdots, B_n\}$,
- neg(r) = $\{B_{n+1}, B_{n+2}, \cdots, B_{n+m}\}$,
- props(r) = $\{$**not** $H\} \cup$ pos(r) $\cup$ neg(r), and
- body(r) = pos(r) $\cup$ $\{$**not** p $\mid$ p $\in$ neg(r)$\}$.

All other rules are in P, and therefore follow our previous definition.

We have defined the language, and can now define what a semantics is. A semantics can be viewed as a function that maps programs to sets of models, and we will use this definition throughout this paper.

Definition 2.7. A semantics, $\mathcal{S}$, is a function mapping programs to sets of models. If for some model M and some program P, $M \in \mathcal{S}(P)$ then we say that M is a model of P with respect to $\mathcal{S}$.

2.2 Coinduction

Our approach is based on coinductive logic programming. Coinductive logic programming is based on the concept of coinduction (the dual of induction) from category theory [8]. Category theory is an abstraction of mathematical studies such a groups and rings [11]. There is a tutorial paper by Jacobs and Rutten that gives a nice introduction to the concept of induction and coinduction without going too heavily into category theory [10].

2.3 SLD & CoSLD Resolution

SLD can be viewed as an inductive proof method based on resolution theory. CoSLD[16, 19] is likewise a coinductive proof method and can be considered a form of circular coinduction[18]. It is our observation that the non-monotonic completion semantics require a combination of induction and coinduction. In [13] a modified CoSLD resolution algorithm is presented in order to allow for induction aspects of stable-model semantics.

This modification uses the standard CoSLD resolution to detect cycles during execution, and decides to succeed or fail based on what is correct for the stable-model semantics. Due to space constraints this paper will not go into details, but they can be found in the original paper, [13].

2.4 The Semantics

This paper divides cycles into three types: positive, even, and odd. Positive cycles contain no negations, odd and even cycles contain an odd and even number of negations, respectively. We will take a look at some semantics that have been used. We will present a brief review of how models are computed, and then discuss how the cycles are handled in each case. For this section we will be using the traditional definition of an interpretation. That is, we will assume interpretations are sets of propositions with the assumption that any missing propositions are false. Starting in section 3 we will use a different definition.

2.4.1 Fitting's 3-Value. Fitting's 3-value semantics[5] was a way to compute the value of predicates(or in our case propositions) that were locally stratified but in a program that was not stratified. Essentially, Fitting's 3-value semantics solves the problem by assigning $\perp$ (Unknown) to any proposition in a cycle. Due to the complexity of the computation method it is not formally reviewed in this paper, but can be found in the original paper [5].

2.4.2 Well-Founded. Well-founded semantics solve the same problem as Fitting's 3-value, but to agree with traditional horn programs it handles positive cycles differently[20].

Definition 2.8. For some program P with interpretation I , $A \subseteq$ lit (P) is an *unfounded set* with respect to I if for all p $\in A$ and all rules, R, of P with p as the head, at least on of the following holds:

- Some literal in the body of R is false in I,
- Some positive literal in the body of R is in A.

Definition 2.9. $\mathbf{U}_P(I)$, the union of all unfounded sets for P with respect to I, is called the *greatest unfounded set* of P with respect to I.

Definition 2.10. Let $\mathbf{W}_P(I) = T_P(I) \cup \neg \cdot \mathbf{U}_P(I)$. Then, the least fixed-point of $\mathbf{W}_P$ is the *well-founded partial model* of P.

If a proposition is in a positive cycle it will be in the greatest unfounded set, and thus assigned the value false. If the value of a proposition depends on a cycle containing a negation, it will not appear in the partial model(and thus assigned $\perp$). It can be seen that neither T_p nor U_p will add the proposition(as a positive or negative literal) to the model.

2.4.3 Stable Models. Stable models uses multiple worlds, rather than assign $\perp$, to stratify the program[6].

Definition 2.11. Let P be a program, and I be an interpretation. The *residual program* of P is the horn logic program computed by the Gelfond-Lifshitz transformation as follows:

- for all propositions p $\in I$ and rules in P, R, remove R if **not** p is in the body.
- remove all negative literals from the resulting program.

Definition 2.12. Let P be a program, and $I \subseteq$ props(P). Then I is a *stable model* if and only if I is the least-fixed point of the residual program for P and I.

If a positive cycle exists in the program, and the truth value of the propositions in the cycle depend only on that cycle then the least fixed-point of the residual program will not contain those propositions. Thus, positive cycles in stable-model Semantics are resolved by assigning false to all propositions in the cycle. For even cycles, two worlds are created. One world for each possible assignment of truth values. For odd cycles if the value a proposition depends on its negation, no model will be found. This can be seen by looking at two different cases.

In the first case we guess p is true. All rules containing **not** p will be removed in the residual program. Since p depended on its negation and the rule that it depend on was remove, p will be false in the least-fixed point of the residual program. Thus, p cannot be true in any model.

For the second case we guess p is false. Since p depends on its negation there exists a rule with **not** p in the body with all other literals in the body being in the least-fixed point of the residual program, and p is in the least-fixed point of the residual program if and only if the body of that rule is true. If this were not the case then the value of p could not depend on its negation. But, if it is the case, **not** p will be removed from the rule, and since all other literals are in the least-fixed point then p must be in the least-fixed point. This does not match our guess, so no model can assign false to p.

2.4.4 CoStable Models. CoStable models is a semantics based on stable models presented in the Co-LP 2016 workshop.[7]

Definition 2.13. The co-residual program of a program P for an interpretation I is computed by the following steps:

- for all propositions p $\in I$ and rules $R \in P$, remove R if **not** p is in the body.
- for all propositions p $\notin I$ and rules $R \in P$, remove R if p is in the body.

```
p :- q.
q :- p.
r :- r.
```

Program 2. Positive Cycle

- Remove all literals from the body of the rules in the resulting program.

Definition 2.14. For some program P, a set of proposition $I \subseteq \text{props}(P)$ is a *costable model* of P if and only if I is the least fixed-point of the coresidual program of P and I.

Costable model semantics is similar to stable model semantics except on how it handles positive cycles. It uses multiple worlds to allow a positive cycle to be true or false. If a set of propositions do not contain any of the propositions that are part of a positive cycle then all rules that form that cycle will be removed from the coresidual program and all such propositions will not be in the least model of the coresidual program. On the other hand if all of the propositions in a positive cycle is in the set then they will have facts in the coresidual program.

We can divide all the propositions in an even cycle into two sets A and B such that $\text{p} \in A$ if there exists $\text{q} \in B$ such that **not** q is in the body of the rule part of the even cycle with p as the head or there exists some $\text{q} \in A$ that is in the body of the rule part of the even cycle with p as the head. In addition we can interchange A and B and the property still holds. By choosing A or B to be a subset of the costable model candidate we can create multiple worlds one where one half is true and one where the other half is true.

Finally, there can be no odd cycles just like stable models.

3 COINDUCTIVE FORMALIZATION

In the previous sections we have claimed that negation-as-failure semantics can be differentiated by how they handle cycles, and that this can be done with a combination of induction and coinduction. We informally showed how this works for several semantics. In this section we will formally define and prove this claim. We will be restricting ourselves to 2 and 3-value logics; treating 2-value logics as a special case of 3-value logics. We will assume that all semantics are completion semantics, and that they do not make use of "special" propositions or meta-logical features. It is our belief that these restrictions could be lifted, but they would complicate the presentation. Therefore, we consider them out of the scope of this paper. One final restriction we will place is on programs. Each program must be *finitely computable*. This will be formally defined later in this section, but informally, a program is finitely computable if there is no way to prove a proposition or its negation using an infinite number of propositions.

For the rest of this paper, we will be using 3-value interpretations where true and false are stated explicitly and $\bot$ is assumed when the proposition is not mentioned.

Definition 3.1. For some program P, a set of literals, $I \subseteq \text{lit}(P)$, is called an *interpretation* for P, and for each proposition, p:

- if $\text{p} \in I$ and **not** $\text{p} \notin I$ then p is true in I.
- if **not** $\text{p} \in I$ and $\text{p} \notin I$ then p is false in I.
- if $\text{p}, \textbf{not}\ \text{p} \notin I$ then p is unknown (or $\bot$) in I.
- if $\text{p}, \textbf{not}\ \text{p} \in I$ then p is said to be *unresolved* in I.

As can be seen from the definition above, a 2-value interpretation is merely an interpretation for which for every proposition p referenced by P, either p or **not** p is in the interpretation.

For an interpretation to be a model of a program it cannot contradict the rules of that program. A literal, when added to an interpretation, that does not cause such a contradiction will be referred to as being supported by that interpretation. This will be a simple, but important concept when proving properties about models.

Definition 3.2. For some program P, literal L is *supported by* interpretation I if and only if there exists some rule in $\text{ext}(P)$ such that:

- L is the head of the rule, and
- for all L' in the body, $L' \in I$, **not** $L' \notin I$.

Example 3.3 (Program 2). For the interpretation $I = \{\text{p}, \text{q}\}$, p is supported by I, but r is not.

Definition 3.4. For some program P, literal L is *supported as unknown by* interpretation I if and only if there exists some rule in $\text{ext}(P)$ such that:

- L is the head of the rule,
- for all L' in the body, $L' \in I$, **not** $L' \notin I$ or L', **not** $L' \notin I$, and
- for at least one literal L' in the body, L', **not** $L' \notin I$.

Example 3.5 (Program 2). Continuing from the last example with interpretation $I = \{\text{p}, \text{q}\}$, r is not supported, but it is supported as unknown.

Since we will view semantics as a mixture of inductive and coinductive semantics, we first need to define what it means to be inductive or coinductive. This is done by constructing a representation of a proof for a literal. If the literal has an inductive proof then it is inductive. If it has a coinductive proof and neither it nor its negation has an inductive proof, it is coinductive. A literal that has neither is not true in any model.

We will represent inductive and coinductive proofs with tree structures. An inductive proof is a tree structure where each node is associated with a literal, and represents a rule in an extended program. The root represents the head, and the children represent the body. Each child is, itself, the root of an inductive proof.

Definition 3.6. Let P be a normal logic program, and L be a literal in $\text{lit}(P)$. Then, L is said to be *inductive* if and only if there exists an *inductive proof* Π_L such that:

- If there exists a fact for L in $\text{ext}(P)$, then Π_L contains a single node with label L.
- Otherwise, if there exists a rule in $\text{ext}(P)$ with L as the head and body $L_1, L_2, \ldots, L_n$ for some $n > 0$ such that $L_1, L_2, \ldots, L_n$ have inductive proofs $\Pi_{L_1}, \Pi_{L_2}, \ldots, \Pi_{L_n}$, respectively, then Π_L is defined by a root node with label L and the roots of $\Pi_{L_1}, \Pi_{L_2}, \ldots, \Pi_{L_n}$ as children.

If a proposition is inductive, then it must be true. If a proposition's negation is inductive then that proposition must always be false. This is because the dual rule can be true only if there is no way to make the proposition true.

Theorem 3.7. *Let P be a program. All inductive literals of P are in all models of the completion of P.*

Proof. Let P be a program and L be an inductive literal with respect to. P. We want to prove that for all models M of P, $L \in M$. We will prove this by induction. It is also important to note that the only assumptions we have made is that L is inductive and that M is a model.

Since L is inductive it must have an inductive proof Π_L.

Base Case: Suppose the height of $\Pi_L = 1$. Then Π_L contains a single node and there must be a fact for L in $\text{ext}(P)$.

- If L is a positive literal, then there exists a clause in the Clark's Completion of P, $L \iff B_1 \vee B_2 \vee \ldots \vee B_n \vee True$ for some $n \geq 0$. This implies $L \iff True$, and thus $L \in M$.
- Otherwise, if L is negative, then there does not exist a rule in P with prop(L) as the head. By the closed world assumption, $L \in M$.

Inductive Hypothesis: Let $k \geq 1$. Assume that for some literal L' if the height of its inductive proof $\Pi_{L'}$ is less than or equal to k then $L' \in M$.

Inductive Step: Assume the height of Π_L is $k+1$. Then there exists a rule in ext(P) s.t. L is the head, and every body literal L' has an inductive proof with height less than or equal to k. By the inductive hypothesis, all such L' are in M. There is a rule in the completion of P prop(L) $\iff B_1 \vee B_2 \vee \ldots \vee B_n$ for some $n > 0$. If L is a positive literal then the body represented by the inductive proof, B_i for some $0 < i \leq n$, must be true and thus $L \in M$. Otherwise, by the definition of extended programs, since the body of a rule for L is true(that is all $L' \in M$), the B_i must be false for all $0 < i \leq n$, and prop(L) must be false. Hence, $L \in M$.

Therefore, by induction, all inductive literals of P are in all models of its completion.

□

A coinductive proof can be viewed as tree structure with infinitely long branches. Each coinductive proof has a literal and a truth value associated with it. Each child can be viewed as a root of a subtree that is, itself, a coinductive proof. As shown above, if a literal has an inductive proof then it will be true in all models. In addition, since we are interested in differentiating between sets of preferred models, we do not need the inductive proofs to be part of the structure, as long as they exist.

Definition 3.8. A coinductive proof c has the following structure:

- the *root* of c, root(c), is the literal being proved,
- the *label* of c, label(c) $\in \{\text{true}, \bot\}$, is the truth value of the root, and
- the *support set* of c, support(c), is a non-empty set of coinductive proofs.

For a literal L of some program P:

- root(c)=L,
- for some rule R in ext(P) with L as the head, for all literals L' in the body, L' is either inductive or has a coinductive proof $c' \in$ support(c), and
- the conjunction of the labels of all coinductive proofs in support(c) is equal to label(c).

For convenience, rule(c) is the rule used to construct the coinductive proof c.

A literal is coinductive if we cannot prove or disprove it inductively. All such literals must be the root of some coinductive proof. This is simply because the literals must be dependent on a cycle or an infinite chain of literals making it impossible to construct an inductive proof.

Definition 3.9. A literal L of a program P, is called *coinductive* if and only if neither L nor its negation are inductive.

Theorem 3.10. *Let P be a program and $L \in$ lit (P) be coinductive. There exists a coinductive proof c such that root(c) = L.*

Proof. First notice that for some program P, if a literal $L \in$ lit (P) is coinductive then

(1) for all rules $r \in$ ext(P) with head(r) = L there exists some $L' \in$ body(r) such that L' is not inductive, and

(2) there exists a rule $r \in \text{ext}(P)$ with $\text{head}(r) = L$ such that all literals in $\text{body}(r)$ are inductive or coinductive.

If property 1 was not true then there was a way to construct an inductive proof, which contradicts the fact that L is coinductive. If the second property was not true then all rules would contain a literal that is not inductive or coinductive. By definition, the negation of such literal must be inductive, and therefore, by the definition of dual rules, the negation of L must be inductive. This contradicts the assumption that L is coinductive. Therefore the properties must hold.

Let L be some coinductive literal in P, Then, let X, called a *rule sequence set*, be a set with each member being an infinite sequence of rules in $\text{ext}(P)$ with the following properties:

- for each rule in each sequence, the head of that rule is coinductive and is in the body of the preceding rule,
- with R being the set of rules used in the sequences in X, $\forall r_1, r_2 \in R : \text{head}(r_1) = \text{head}(r_2) \Rightarrow r_1 = r_2$,
- for all sequences in X, for the first rule, r,
  - $\text{head}(r) = L$,
  - with C being the set of all coinductive literals in $\text{body}(r)$ and R being the set of all rules that is the second rule in some sequence in X, $\exists L' \in C \iff \exists r' \in R : L' \in \text{body}(r')$, and
  - for each set X' such that $s \in X'$,with the first rule of s being r', if and only if s', constructed by prepending r to s is in $\{s'' \mid s'' \in X, \text{Second rule of } s'' \text{ is } r'\}$, X' a rule sequence set.

Now we must show that X cannot be empty. Let $\mathbb{L}$ be the set of all infinite sequences of rules in $\text{ext}(P)$ starting with some rule $r \in \text{ext}(P)$ such that $\text{head}(r) = L$, such that for each rule in each sequance, the head of that rule is coinductive and is in the body of the preceding rule. If $\mathbb{L}$ is empty, there there must be some coinductive proof without a rule. This violates Property 2 of coinductive literals. So, $\mathbb{L}$ cannot be empty. Now, let $\mathbb{L}_2$ be a subset of $\mathbb{L}$ such that with R being the set of rules used in the sequences in $\mathbb{L}_2$, $\forall r_1, r_2 \in R : \text{head}(r_1) = \text{head}(r_2) \Rightarrow r_1 = r_2$, and for all $s \in \mathbb{L}$ $\{s\} \cup \mathbb{L}_2$ does not satisfy this condition. Since it is suffiecient to have a single true rule to prove a literal, if a rule appears in a sequence it can be used anytime that literal is in the body of the preceding rule, and thus $\mathbb{L}_2$ cannot be empty. Since we cannot add another sequence to $\mathbb{L}_2$ without introducing a new rule that has the same head as one already used is some sequence then the forth property for rule sequence sets must be satisfied. Furthermore, this condition also ensures this property for all rules in all sequences in $\mathbb{L}_2$. Finally, for each set $\mathbb{L}_2'$ such that $s \in \mathbb{L}_2'$ if and only if s', constructed by prepending r to s is in $\mathbb{L}_2$, $\mathbb{L}_2'$ satisfies the properties to be a rule sequence set. Therefore, a rule sequence set for a coinductive literal cannot be empty.

We have show that there is a nonempty rule sequence set for any coinductive literal. Let X be a rule sequence set for some coinductive literal L. We will construct a coinductive proof, c, as follows:

- $\text{root}(c) = L$,
- $\text{label}(c) = \bot$,
- $\text{rule}(c)$ is the first rule of the sequences in X, and
- $\text{support}(c)$ is the set of coinductive proofs constructed from each rule sequence set X' such that $s \in X'$ if and only if s', constructed by prepending r to s is in X.

Since rule sequence sets have the fourth property it must be the case that for each literal in $\text{body}(\text{rule}(c))$ it is either inductive or has a coinductive proof in $\text{support}(c)$, and since $\bot$ is assigned to each coinductive proof, the conjunction of all labels of $\text{support}(c)$ will be $\bot$ and thus equal to $\text{label}(c)$.

Therefore, for all coinductive literals L, there exists a coinductive proof, c, such that $\text{root}(c) = L$. □

We now have enough information to formally define computability. We will do this by looking at how many literals are needed to form the coinductive proofs. If there is a finite number of literals it is finitely computable.

Definition 3.11. Let c be a coinductive proof. Then we can construct a function ϕ_c such that

$$\phi_c(X) = \begin{cases} \{c\} & : \quad X = \emptyset \\ \bigcup\limits_{c' \in X} \text{support}(c') & : \quad \text{otherwise.} \end{cases}$$

The greatest fixed point of ϕ_c is the *literal set* of c.

It can be seen that if the literal set of a coinductive proof is not finite we cannot enumerate the literals in finite time. On the other hand, if it is finite we can. This is exactly what we mean by finitely computable.

Definition 3.12. Let c be a coinductive proof. If the literal set of c has finite cardinality then we say c is *finitely computable*. In addition, for some program P, P is *finitely computable* if no $L \in \text{lit}(P)$ has a coinductive proof that is not finitely computable.

It is possible to compute a model of a program that has a literal with a coinductive proof that is not finitely computable as long as that coinductive proof is not needed for any model. But, this is the same as transforming the program by removing rules that lead to it not being finitely computable. As stated earlier in this section, this paper will assume all programs will be finitely computable.

To represent a model of a program we need only to keep track of the coinductive proofs. As with the definition of coinductive proofs it is enough to know the inductive proofs exist, and a set of coinductive proofs is all that is needed to differentiate between models. This set will need to adhere to several properties to be recognised as a model.

Definition 3.13. For a program P, a set of coinductive proofs, C, is called a *coinductive proof set*, and defines two sets:

- $\text{R}(C)=\{(L,T) | c \in C, L = \text{root}(c), T = \text{label}(c)\}$
- $\text{Sup}(C)= \bigcup\limits_{c \in C} \text{support}(c)$.

For a coinductive proof set to be a model it cannot depend on assigning different values to the same proposition. We call such a set consistent.

Definition 3.14. Let c and c' be coinductive proofs for some program. c' *contradicts* c if and only if:

- $\text{root}(c) = \text{root}(c')$ and $\text{label}(c) \neq \text{label}(c')$,
- $\text{root}(c)$ is the negation of $\text{root}(c')$ and $\text{label}(c) \neq \textbf{not}\ \text{label}(c')$, or
- $\exists c'' \in \text{support}(c)$ such that c' contradicts c''.

For some program P, a coinductive proof set C is *consistent* if and only if $\forall c, c' \in C$ c does not contradict c'. We say that C is *inconsistent* if it is not consistent.

For a single program there may be multiple ways to assign the same value to a literal. For instance there may be multiple rules for a literal or a literal in the body of the rule may have multiple rules. Since we are concerned with the value each literal will ultimately be assigned it does not matter which path is used to prove a literal has a certain value. All such proofs are *equivalent* and should not be considered different proofs with respect to coinductive proof sets.

Definition 3.15. Let c_1, c_2 be coinductive proofs. We say c_1 and c_2 are *equivalent* if and only if $\text{root}(c_1) = \text{root}(c_2)$ and $\text{label}(c_1) = \text{label}(c_2)$. Furthermore, a coinductive proof set C *covers* a coinductive proof c_1 if and only if there is some $c_2 \in C$ that is equivalent to c_1.

If, for some coinductive proof sets C_1 and C_2, C_1 covers all members of C_2 and C_2 covers all members of C_1 then C_1 and C_2 are *equivalent*($C_1 \equiv C_2$).

For convenience we want every coinductive literal that is assigned true or $\bot$ to have a coinductive proof in the set. This ensures that a coinductive proof for a literal is readily available, and we do not have to go deeply into the structure. Thus simplifying proofs.

Definition 3.16. A coinductive proof set C for some program P is called *complete* if and only if for all propositions $p \in \text{lit}(P)$ if p is coinductive, then C covers p or **not** p.

If a literal has a value of $\bot$ then it's negation must also have a value of $\bot$. A complete coinductive proof set does not guarantee such a coinductive proof will be in the set. However, if there is such a coinductive proof set then there exists a superset that contains a coinductive proof with the literals negation as the root. Therefore, we will ignore these sets and only take the sets with no such superset.

Definition 3.17. Let C_1 and C_2 be coinductive proof sets. We say C_2 is *larger* than C_1 if

- there exists some $c \in C_2$ such that C_1 does not cover c, but $\text{Sup}(C_1)$ covers c, or
- C_2 is larger than $C_1 \cup \text{Sup}(C_1)$.

If, for some coinductive proof set C, there does not exist some other coinductive proof set C' such that C' is larger than C then C is said to be a *largest* coinductive proof.

If a literal has multiple coinductive proofs constructed from different rules the actual value of the literal is a disjunction of the labels. When all the labels are true this could be ignored, but it is possible for some of the labels to be $\bot$. In this case these coinductive proofs cannot be used in a model. We could never assign $\bot$ to a literal if we have a rule that makes it true. As an example consider the following program when viewed by well-founded semantics.

```
p :- p.
q :- not r.
r :- not q.
s :- not p.
s :- q.
```

Since p is unfounded it will be assigned false , and therefore **not** p must be true. Since q is neither founded nor unfounded it will be assigned $\bot$. That is there exists a coinductive proof for s from its first rule that has a true label and one form its second rule with a $\bot$ label. For the coinductive proof set to correspond to a model, the coinductive proof from the second rule cannot be used. We call such coinductive proofs *invalid.*

It will be convenient to recognize when a coinductive proof set will make a rule for a coinductive proof true. This will allow us to identify when a different rule would make a literal true.

Definition 3.18. Let C be a coinductive proof set, and c be a coinductive proof. If for all $c' \in \text{support}(c)$, C covers c' then we say C *supports* c (or c is supported by C).

Definition 3.19. Let C be a complete coinductive proof set. Then, C is *invalid* if and only if there exists $c \in C$ with $\text{label}(c) = \bot$ and there exists a coinductive proof c' supported by C such that $\text{label}(c') = \text{true}$ and either $\text{root}(c')$ or **not** $\text{root}(c')$ is $\text{root}(c)$.

If C is not invalid, we say it is *valid.*

All these properties must be true for a coinductive proof to correspond to a model. We call such sets proof models.

Definition 3.20. For some program P, a *proof model* of P is a largest coinductive proof set for P that is complete, consistent, and valid.

Lemma 3.21. *Let P be a program, and I an interpretation for P with no unresolved propositions. I is a model for the completion of P if*

(1) $\forall L \in M$, *L is supported by M.*
(2) $\forall L \in$ *lit* (P) *such that* L, ***not*** $L \notin M$, *L is supported with unknown by M.*

Proof. Asume properties 1 and 2 hold for I, but I is not a model. Then there exists some rule $h \iff B_1 \vee B_2 \vee \ldots \vee B_n$ for some $n > 0$ that is false. There are three cases:

h ∈ I. In this case, h is true and h is supported by I. Therefore there exists some i such that B_i is true. Therefore, $h \iff B_1 \vee B_2 \vee \ldots \vee B_n$ must be true. A contradiction.

not h ∈ I. In this case, h is false and **not** h is supported by I. By the definition of "supported by" and dual rules, for all $0 < i \leq n$, there exists a literal L_i in B_i that is false. Therefore, all B_i are false, and $h \iff B_1 \vee B_2 \vee \ldots \vee B_n$ is true.

h, not h ∉ I. In this case, h is $\perp$ and is supported with unknown by I. By the definition of "supported with unknown by", for all $0 < i \leq n$, there exists a literal L_i in B_i that is $\perp$ and all literals in B_i must be true or $\perp$. Therefore, all B_i are $\perp$, and $h \iff B_1 \vee B_2 \vee \ldots \vee B_n$ is true.

□

Theorem 3.22. *Let P be a program. The set of all models for P is equivalent to the set of all proof models of P.*

Proof.

Case 1: Suppose C is a proof model of P. We wish to show that there exists a corresponding model.

Since C is complete and a largest coinductive proof set, every coinductive literal is represented in R(C). Now, let A be the set of all inductive literals in ext(P), $B = \{L \mid (L, \text{true}) \in \text{R}(C)\}$, and $M = A \cup B$. From lemma 3.21, to show M is a model of P we must show:

(1) $\forall L \in M$, L is supported by M.
(2) $\forall L \in$ lit (P) such that L, **not** $L \notin M$, L is supported with unknown by M.

Property 1: If L is inductive, then either there is a fact for L(and therefore supported) or there exists a rule with L as the head such that each body literal L' is inductive. Thus, each L' is in A(and therefore in M), and L is supported.
If L is not inductive then it must be in B. So, there exists a coinductive proof $c \in C$ with root(c)=L, label (c)=true , and $\forall c' \in$ support(c), label(c')=true. Since C is consistent, for all such c', label (c') is in M, and since any inductive literals in the body will be in A we can say that L is supported.

Property 2: Since both L and **not** L are not in M we know that neither are inductive, and must be coinductive, since C is complete. There exists a $c \in C$ with root(c) = L and label $(c) = \perp$. Since c was constructed from a rule with L as the head and each body literal L' being either inductive(and thus $L' \in A$), coinductive and true(and thus $L' \in B$), or coinductive and unknown(and thus both L', **not** $L' \notin M$). Thus L is supported with unknown.

Case 2 Let M be a model of P. We wish to show there is a corresponding proof model.

We know that for all $L \in M$ there exists a rule with L as the head and for all literals L' in the body, $L' \in M$. For each $L \in M$ such that L and L' are both not inductive we can construct a coinductive proof with L as the root, true as the label, and the support set made from the coinductive literals in the body of the rule. Of which, there must be at least one since L is not inductive.

In addition, for all literals L such that L, **not** $L \notin M$ there exists a rule with L as the head and for all L' in the body $L' \in M$ or L', **not** $L' \notin M$(and there is at least one). So, we may construct a coinductive proof with L as the root, $\perp$ as the label, and the support set made from the coinductive proofs of all coinductive literals of the body(of which there is at least one since L must be coinductive since M is a model).

It is easy to see that the set containing all such coinductive proofs is complete since there is a proof for each proposition p such that p, **not** p are both not inductive. It is a largest coinductive proof set, since there is a coinductive proof for each literal L with L, **not** $L \notin M$. It is consistent since if a coinductive proof contradicts another then that contradiction will be in the rules used to construct the coinductive proofs, which violates the assumption that M is a model. And finally, it must be valid otherwise there would be a rule that must assign true to some L such that L, **not** $L \notin M$, and therefore M couldn't be a model.

□

We can extract a cycle by following a branch of the coinductive proof tree. Eventually we will reach a literal we have seen before and we know that is a cycle. However, since there can be more than one rule that can be used, it is possible that a proposition could be repeated multiple times before a cycle forms. Therefore, we want to restrict the number of rules used to a minimum.

Definition 3.23. Let c_1, c_2 be equivalent coinductive proofs. Let A_1, A_2 be the literal sets of c_1 and c_2, respectively. We say c_1 is *simpler* than c_2 if $|A_1| < |A_2|$.

Definition 3.24. Let c be a coinductive proof of some program P. If there does not exist a coinductive proof of P, c' such that c' is simpler than c. Then c is said to be *simplest*.

Definition 3.25. Let C be a proof model for some program P. Then C is said to be the *simplest equivalent* proof model if there does not exist a proof model C' where $C' \equiv C$, and $\exists c \in C$ and $\exists c' \in C'$ such that c' is simpler than c.

Since we will be differentiating semantics based on the how they handle cycles we need a formal definition of what a cycle is. For this paper a cycle will refer to the circumstance where a proposition's truth value depends on itself.

Definition 3.26. Let c be a coinductive proof, and $L_0, L_1, L_2, \ldots, L_n$ for some $n > 0$ be a sequence of literals. Then, $L_0, L_1, L_2, \ldots, L_n$ is called a *direct cycle* of c if $\text{root}(c) = L_0$ and $\exists c' \in \text{support}(c)$ such that $L_1, L_2, \ldots, L_n, L_0$ is a direct cycle of c'.

There is an associated value of $\text{label}(c)$.

Definition 3.27. Let c be a coinductive proof, and $L_0, L_1, L_2, \ldots, L_n$ for some $n > 0$ be a sequence of literals. Then, $L_0, L_1, L_2, \ldots, L_n$ is called a *indirect cycle* of c if $L_0, L_1, \ldots, L_n$ is not a direct cycle of c and

- $\exists c' \in \text{support}(c)$ such that $L_0, L_1, L_2, \ldots, L_n$ is a direct cycle of c, or
- $\exists c' \in \text{support}(c)$ such that $L_0, L_1, L_2, \ldots, L_n$ is an indirect cycle of c'.

Definition 3.28. Let c be a coinductive proof, and $L_0, L_1, L_2, \ldots, L_n$ for some $n > 0$ be a sequence of literals. Then, $L_0, L_1, L_2, \ldots, L_n$ is called a *cycle* of c if it is either a direct cycle or an indirect cycle of c.

In addition, $L_n, L_0, L_1, \ldots, L_{n-1}$ and all $L_i, \ldots, L_n, L_0, \ldots, L_{i-1}$, with $0 < i < n$, are *equivalent* to $L_0, L_1, \ldots, L_n$.

We have claimed that all semantics within the restrictions differentiate models based on the cycles. This point is further emphasized by the fact that the labels for coinductive proofs that do not have a direct cycle depend only on the coinductive proofs in the indirect cycles.

Lemma 3.29. *For some program P, let c be a valid coinductive proof with no direct cycles. Let X be the set of coinductive proofs that have some indirect cycle for c as a direct cycle. Let C_1 and C_2 be proof models that are not equivalent such that both cover X. Both C_1 and C_2 cover c.*

Proof. This claim can be shown by inducting on the level of indirectness. The level of a coinductive proof that has no indirect cycles is 0. All other coinductive proofs have a level of one greater than the highest leveled coinductive proof in its support set.

Base Case. Suppose c has a level of 1. Since, X must support c, C_1 and C_2 must support c. Since c is valid, both C_1 and C_2 must cover c.

Inductive Hypothosis. Assume for some integer k, all coinductive proofs c' with level less than or equal to k are covered by C_1 and C_2 if C_1 and C_2 cover some coinductive proof set X' that contain the coinductive proofs with some indirect cycle from c' as a direct cycle.

Inductive Step. Assume c has level $k + 1$. Then, all coinductive proofs in support(c) must have level k or less. By the inductive hypothesis, these coinductive proofs must be covered by C_1 and C_2, and X is the union of all X' from support(c). Since c is valid, and C_1 and C_2 support c, C_1 and C_2 must cover c.

By induction, c is covered by C_1 and C_2. □

A cycle is resolved by assigning the same truth value to all literals in the cycle. We categorise cycles into three types.

Definition 3.30. Let $L_0, L_1, L_2, \ldots, L_n$ with $n \geq 0$ be a cycle for some simplest coinductive proof c.

- If $L_0, L_1, L_2, \ldots, L_n$ are all positive literals or are all negative literals, then the cycle is a *positive cycle.*
- If $\exists i, j \leq n$ such that $L_i = \textbf{not}\ L_j$, then the cycle is called an *odd cycle.*
- Otherwise the cycle is an *even cycle*

Now we can present an algorithm to represent a semantics based on how the cycles are resolved. To do this we will "filter" the set of all possible models for a program, leaving only the preferred models. For each semantics, three functions must be defined. Each function maps a cycle, proof model, and program to a set of truth values. These three functions corrispond to the three cycle types. Since the function can depend on the proof model or even the program as a whole, the accepted truth values of a cycle can depend on global information, locally seeming like an exception.

Informally, to compute the subset of proof models we take the set of all proof models and for each proof model we check each of its cycles. Each cycle can be said to have a truth value associated with them based on the labels of the coinductive proofs associated with it. If there is a cycle for which its truth value does not match the value assigned by the corresponding function, then the proof model is removed from the set. The result is the set of all proof models with respect to the semantics.

There is, of course, a problem with the simplistic approach above. It can be illustrated by a simple example.

```
p :- p, not q.
q :- not p.
```

Suppose the function for positive cycles always associates true , and the function for even cycles always associates $\bot$. The intended preferred model for this program would be for both p and q to be $\bot$. However, the positive cycle will have $\bot$ associated with it from the coinductive proof, and therefore not match a value from the function for positive cycles. This results in being no preferred models.

The problem above is because the positive cycle is a direct cycle for p and the truth value is determined by the conjunction of the truth values of the two cycles. To solve this we need to change the algorithm slightly to account for this. The truth value of coinductive literals depend entirely on their cycles, and if we expand by replacing the current coinductive proof with its support set we will see that the truth value is the conjunction of all cycles.

Definition 3.31. Let f_p, f_e, f_o be functions that maps a cycle (positive, even and odd cycles, respectively), a coinductive proof, and a program to a subset of {true , $\bot$ }. Let C be a proof model for some program P. Let f be a function such that:

$$f(x,c,p) = \begin{cases} f_p(x,c,p) & \text{if } x \text{ is a positive cycle} \\ f_e(x,c,p) & \text{if } x \text{ is a even cycle} \\ f_o(x,c,p) & \text{if } x \text{ is a odd cycle} \end{cases}$$

We say that C is *accepted* by f_p, f_e, f_o for P if and only if $\forall c \in C$, with A being the set of all cycles of c, there exists function $\tau : A \rightarrow$ {true , $\bot$ } such that $\bigwedge_{a \in A} (\tau(a) \in f(a,c,P) \wedge \tau(a)) = \text{label}(c)$.

If C' is a set of proof models, then it is accepted for some program P if and only if: $c \in C' \iff$ c is accepted for P.

Now we can create a function that returns all accepted proof models for a program. Since eachproof model can be directly converted to a model, this function can be considered a semantics.

Definition 3.32. Let f_p, f_e, f_o be functions that maps cycles (positive, even and odd cycles, respectively), a coinductive proof, and a program to a subset of {true , $\bot$ } such that equivalent cycles are mapped to the same set. Let $\mathcal{P}$ be a function such that for all programs P, $\mathcal{P}(P)$ is the set of proof models accepted by f_p, f_e, f_o for P. $\mathcal{P}$ is called a *proof model function.*

If, for some semantics $\mathcal{S}$ and all programs P, $\mathcal{S}(P) = \{M \mid M' \in \mathcal{P}(P), M = \text{model}(M)\}$ then we say that $\mathcal{P}$ is the *proof model form* of $\mathcal{S}$.

Theorem 3.33. *Let $\mathcal{P}$ be a proof model function. There exists a semantics, $\mathcal{S}$, such that for all programs P, $\mathcal{P}(P)$ is equivalent to the set of all models with respect to $\mathcal{S}$.*

Proof. From lemma 3.22, we know that all proof models can be converted to a model for P. Let $\mathcal{M}$ be a function that takes a set of proof models and converts it to a set of models. Now, let $\mathcal{S}(P) = \mathcal{M}(\mathcal{P}(P))$. It is clear that the result will be a set of models that is equivalent to $\mathcal{P}(P)$. □

Now to prepare to prove every semantics has a proof model form we need to show directly how cycles affect the label of a coinductive proof. We want to show that the label of a coinductive proof is the conjunction of the associated value of all of its cycles. This works because we are assuming we will be working with only finitely computable programs.

Lemma 3.34. *Let c be a finitely computable coinductive proof and A be the set of all cycles for c. For all $a \in A$ let $\tau(a)$ be the associated value for a in c. Then, $\bigwedge_{a \in A} \tau(a) = \text{label}(c)$.*

Proof. Since c is finitely computable we know that there is a finite number of literals used to construct it. So we can prove our claim by inducting on the level of indirectness as in lemma 3.29.

Base Case. Suppose the level of indirectness for c is zero. That is there are no indirect cycles. Then, by defintion the associated values of all cycles is $\text{label}(c)$. So,

$$\bigwedge_{a \in A} \tau(a) = \bigwedge_{a \in A} \text{label}(c) = \text{label}(c).$$

Inductive Hypothesis. Let c' be a coinductive proof with a level of indirectness less than or equal to k, A' be the set of all cycles for c', and for all $a \in A'$, $\tau_{c'}(a)$ is the associated value for a in c'. Assume $\bigwedge_{a \in A'} \tau_{c'}(a) = \text{label}(c')$.

Inductive Step. Suppose c has a level of indirectness of $k+1$. Now for all $c' \in \text{support}(c)$, c' has a level of indirectness of k or less and, by the inductive hypothesis, $\bigwedge_{a \in A_{c'}} \tau_{c'}(a) = \text{label}(c')$. From the definition of a cycle it follows that $A = \bigcup_{c' \in \text{support}(c)} A_{c'}$, and by defintion(of coinductive proofs)

$$\text{label}(c) = \bigwedge\nolimits_{c' \in \text{support}(c)} \text{label}(c') = \bigwedge\nolimits_{c' \in \text{support}(c)} \left(\bigwedge_{a \in A_{c'}} \tau_{c'}(a) \right) = \bigwedge_{a \in A} \tau(a)$$

.

By induction $\bigwedge_{a \in A} \tau(a) = \text{label}(c)$. □

We can construct a proof model form directly from a semantics by assigning values to cycles that match the values assigned to the literals by the models of the semantics.

Theorem 3.35. *Every semantics has a proof model form.*

Proof. Let $\mathcal{S}$ be some completion semantics. We can define the function

$$f_{\mathcal{S}}(C, M, P) = \{ \text{label}(c) \mid L \in C, c \in M, \text{root}(c) = L, \text{model}(M) \in \mathcal{S}(P) \}.$$

Now, $f_p(C, M, P) = f_e(C, M, P) = f_o(C, M, P) = f_{\mathcal{S}}(C, M, P)$ defines a proof model form. Since $f_{\mathcal{S}}$ gives the values that the literals in a cycle can take in any model with respect to $\mathcal{S}$(and therefore a function that maps cycles to truth values can always be constructed), a proof model will be accepted by this proof model form if and only if the equivalent model is a model with respect to $\mathcal{S}$.

To show this, assume that for some program P, the set of accepted proof models for f_p, f_e, f_o is not equivalent to the set of models of $\mathcal{S}$. Then either there is a proof model, M, accepted by f_p, f_e, f_o such that model(M) is not a model with respect to $\mathcal{S}$ or there is a model, M, of $\mathcal{S}$ such that proofmodel(M) is not accepted by f_p, f_e, f_o.

Now, assume there exists a proof model M that is accepted by f_p, f_e, f_o such that model(M) is not a model with respect to $\mathcal{S}$. From theorem 3.22, we know that model(M) is a model of P. Since M is accepted we know that $\forall c \in M$, with A being the set of all cycles of c, there exists function $\tau : A \rightarrow \{\text{true}, \bot\}$ such that $\bigwedge_{a \in A} (\tau(a) \in f(a, c, P) \wedge \tau(a)) = \text{label}(c)$.

Since model(M) is not a model of P with respect to $\mathcal{S}$, $f_{\mathcal{S}}(C, M, P)$ must be empty. But this contradicts the existance of τ. Therefore, if M is accepted by f_p, f_e, f_o it must be the case that model(M) is a model of P with respect to $\mathcal{S}$.

Now, assume that there is a model M of P with respect to $\mathcal{S}$, but proofmodel(M) is not accepted by f_p, f_e, f_o. For all $c \in$ proofmodel(M) we can construct a funtion τ by assigning to each cycle in c the associated values of those cycles. It is clear to see that for all cycles, C, $\tau(C) \in f_{\mathcal{S}}(C, c, P)$. So, the only way for proofmodel(M) to not be accepted is if for all $c \in$ proofmodel(M), $\bigwedge_{a \in A} (\tau(a) \in$

$f(a, c, P) \wedge \tau(a)) \neq \text{label}(c)$. Since for all a, $\tau(a)$ is the associated value in c, by lemma 3.34 this cannot be the case. Therefore, if M is a model of P with respect to $\mathcal{S}$ then proofmodel(M) must be accepted by f_p, f_e, f_o. □

4 FIXED-POINT FORMALIZATION

As with theorem 3.35, we want to represent a semantics in terms of how they treat cycles. To do this we need a way to detect and resolve cycles in an incremental way while computing a fixed-point. The chosen method makes use of what we call cycle sets and cycle resolutions. A cycle set is simply the collection of rules forming a cycle, and a cycle resolution is a set of positive and negative literals that can be subtracted from an interpretation to change the value of the associated propositions from unresolved to some truth value.

Definition 4.1. Let P be a program and C be a nonempty subset of P. C is said to be a *set of cyclic rules* if and only if for all $r \in C$:

- there exists $r' \in C$ such that $\text{head}(r) \in \text{pos}(r') \cup \text{neg}(r')$ and
- there exists $p \in \text{pos}(r) \cup \text{neg}(r)$ and $r' \in C$ such that $\text{head}(r') = p$.

C is a *cycle set* of P if it is a set of cyclic rules and there is no subset of C that is also a set of cyclic rules.

Definition 4.2. Let P be a program, I be an interpretation for P, and C be a cycle set of P. If for all propositions p that is the head of some rule in C, p is unresolved in I then C is said to be a *cycle set with respect to I*.

While cycles (as defined in section 3) represent a cyclic relationship that must be resolved to prove some literal, a cycle set is the set of rules that potentially form a cyclic relationship. There is no guarantee that the rules are truly cyclic. For instance, consider the program containing a single rule "p:− p,q.". This rule does not really form a cycle. Proposition q will be false and therefore p must be false. Later in this section we will present a way of determining if a cycle set is truly cyclic when trying to resolve it.

First, however, we must present some properties of cycle sets, and introduce cycle resolutions and cycle resolution functions.

For any proposition in a cycle set, it or its negation will be in some rule. It is useful to differentiate between these two cases, and we will call them positively and negatively referenced, respectively.

Definition 4.3. Let C be some cycle set. Proposition p is said to be *negatively referenced* if and only if p is the head of some rule in C and there exists some rule $r \in C$ such that $\text{p} \in \text{neg}(r)$.

Definition 4.4. Let C be some cycle set. Then proposition p is said to be *positively referenced* if and only if p is the head of some rule in C and there exists some rule $r \in C$ such that $\text{p} \in \text{pos}(r)$.

When resolving cycles we will first resolve all positive and even cycles first. Remaining unresolved cycles must be odd. So, we will only explicitly deal with positive and even cycles.

Definition 4.5. Let P be a program with interpretation I and C be a cycle set with respect to I. C is called a *positive* cycle set if and only if for all rules $r \in C$, head(r) is positively referenced.

Definition 4.6. Let P be a program with interpretation I and C be a cycle set with respect to I. C is called an *even* cycle set if and only if the number of rules $r \in C$ with head(r) negatively referenced is non-zero and even.

Definition 4.7. Let P be a program with interpretation I and C_1, C_2 be cycle sets with respect to I. If C_1 and C_2 are both positive cycle sets or both even cycle sets, then we say they are the same *type of cycle*.

We have defined how to detect cycles. Now we must have a way to resolve them. Informally, resolving a cycle is merely assigning the propositions in it a value.

Definition 4.8. Let C be a cycle set of some program P, and $R \subseteq \text{lit}(P)$ such that for all $L \in R$ there exists some rule $r \in C$ such that $\text{head}(r) = \text{prop}(L)$, and for all $r' \in C$, $\text{head}(r')$ or its negation is in R. Then, R is called a *cycle resolution* for C. We also say R *resolves* C.

Furthermore $\text{lit}(P) \setminus R$ is an interpretation specifying the value of the propositions referenced in R.

Each cycle paired with a cycle resolution can be associated with a cycle.

Theorem 4.9. *Let C be a cycle set of some program P and R a cycle resolution for C. Then there exists a cycle $L_0, L_1, ..., L_n$ for some $n \geq 0$ of some coinductive proof for P that assigns the same values to the propositions involved.*

Proof. We can build the cycle from C, and R as follows:

- Choose a rule $r_0 \in C$. Then, L_0 is such that **not** $L_0 \in R$ and $\text{head}(r_0) = \text{prop}(L_0)$.
- For $0 < i \leq n$, choose a rule $r_i \in C$ such that $\text{head}(r_i) = \text{prop}(L_i)$, **not** $L_i \in R$, and if L_{i-1} is negated then **not** $L_i \in \text{body}(r_{i-1})$ otherwise $L_i \in \text{body}(r_{i-1})$.

If for all $L \in R$, **not** $L \in R$ then we assign $\bot$ to the cycle, otherwise we assign true. □

We will use cycle sets to generate resolutions, but it is not enough for us to handle only a single cycle. We must take into consideration all possible cycles, and the possibility of multiple worlds. We do this through cycle resolution functions. These functions map a program and interpretation to a set of complete sets of cycle resolutions. Each set of cycle resolutions represent a different world, where the cycles are resolved differently. In the end we will define a semantics in terms of these functions.

Definition 4.10. Let A be a largest set of cycle sets for some program with respect to some interpretation such that all cycle sets in A are the same type of cycle set, and R be a set of cycle resolutions. Then, R is called *complete* if and only if

- $R = \emptyset \Rightarrow A = \emptyset$,
- for all $X, Y \in R$ with $X \neq Y$, X and Y do not resolve the same cycle set, and
- for all $X \in A$ there exists a $Y \in R$ such that Y resolves X.

Definition 4.11. Let R be a set of cycle resolutions and C be a set of cycle sets. Then we say that R *resolves* C if and only if:

- $\forall r \in R, \exists c \in C$ such that r resolves c, and
- $\forall c \in C, \exists r \in R$ such that r resolves c.

Definition 4.12. For some program P, a function that uses the cycle sets of P to map an interpretation to a set of complete sets of cycle resolutions is called a *cycle resolution function.* Furthermore, we call it an *even cycle resolution function* or *positive cycle resolution function* according to the type of cycle the resulting resolutions will resolve.

Since each resolution function is going to need to recognize the set of cycles for a particular interpretation, we will define a function for convenience.

Definition 4.13. Let P be a program, and I an interpretation of P. $\mathcal{C}(I)$ is the largest set of cycle sets such that $\forall C \in \mathcal{C}(I)$, C is a cycle set with respect to I.

Furthermore, $\mathcal{C}^+(I) \subseteq \mathcal{C}(I)$ is the largest subset of positive cycle sets, and $\mathcal{C}^-(I) \subseteq \mathcal{C}(I)$ is the largest subset of even cycle sets.

Our fixed-point formalization will be centered around what we will call resolution form. Like with proof model form, we will define a semantics in terms of how cycles are resolved. In this case we will use cycle resolution functions. It should be noted that cycle resolution functions cannot make use of information not part of the cycle like the functions in proof model form. To counter this we will add two "filter" functions that deal with this information at two different levels. One function will deal with the completed model, and another will deal with the set of all possible models. That is, the interpretation and the program levels. As stated, we will require that all positive and even cycle sets can be resolved with the corresponding functions, and that any unresolved propositions afterwards must be part of an odd cycle. Therefore we will combine the handling of odd cycles with the "interpretation" level filter function.

Definition 4.14. Let PC be a positive cycle resolution function, and EC be an even cycle resolution function. Let L(called a *local filter function*) be a function that maps interpretations to interpretations by either resolving an unresolved proposition or making an resolved proposition unresolved, and G(called a *global filter function*) be a function mapping sets of interpretations to sets of interpretations such that $\mathtt{G}(C) \subseteq C$ for some C. Then, the 4-tuple (PC, EC, L, G) is called a semantics in *resolution form.*

Our goal of this section is to present a function that is parameterized by a semantics in resolution form and a program such that some fixed-point of that function is the set of all models for the program. This means our function, must map sets of interpretations to sets of interpretations, and we need a method to identify the fixed-point that contains the models. We will call this the Herbrand fixed-point.

Definition 4.15. Let P be a program, and I be the set containing lit (P) as its only member. Then for some function F that maps from and to a set of interpretations, $F^{\infty}(I)$ is the *Herbrand Fixed-Point.*

Before we can define the generalized function used to compute the models we need a few more definitions. Firstly, since positive and negative literals behave the same, except in cycles, we will take advantage of the extended program to handle negative information. For this we must extend the traditional T_P operator.

Definition 4.16. The *Extended* $\mathbf{T_P}$ operator $\mathbf{T'_p}$ is defined as follows:
For the interpretation I and program P:

- $L \in \mathbf{T'_p}(I)$ if there is a fact for L in ext(P),
- $L \in \mathbf{T'_p}(I)$ if there exists a rule $r \in$ ext(P) with head$(r) = L$ such that body$(r) \subseteq I$, and
- $L \notin \mathbf{T'_p}(I)$ otherwise.

Lemma 4.17. *Let P be a program, I be some interpretation of P, and L be an inductive literal in* lit (P). *If I is a fixed-point of $\mathbf{T'_p}$, then $L \in \mathbf{T'_p}(I)$.*

Proof. Since L is inductive there must exist an inductive proof π for L. We can induct on the height of π.

Base Case. Assume π has a height of 1. That is, the root of π has no children. Then, there must be a fact in ext(P) for L, and by definition of $\mathbf{T'_p}$, $L \in \mathbf{T'_p}(I)$.

Inductive Hypothesis. Assume for all inductive proofs, π', with height less than or equal to k, root$(\pi') \in \mathbf{T'_p}(I)$.

Inductive Step. Suppose π has a height of $k+1$. Then, we know that all children of the root has a height less than or equal to k. By the inductive hypothesis we know that the roots of those proofs must be in $\mathbf{T'_p}(I)$. Since those literals are the body literals of the rule used to construct π, and I is a fixed-point of $\mathbf{T'_p}$ it must be the case that $L \in \mathbf{T'_p}(I)$.

```
p :- not q, r.
q :- not p.
```

Program 3. False Cycle

□

The $\mathbf{T}'_{\mathbf{p}}$ operator allows us to treat positive and negative literals the same, but does not handle cycles. The approach we will use is to use the $\mathbf{T}'_{\mathbf{p}}$ operator to progress towards the fixed-point, but when we encounter a cycle we will externally resolve it. This is the purpose of the cycle resolution functions, however not all cycle sets resolved by a cycle resolution function are true cycles. To reiterate, consider program 3.

These two rules form a cycle set and there is a resolution for it where p is assigned true. Since r has no rule, however, we know that p must be false. This can be further complicated by adding the rule "r:- p.". Now p is part of two cycles. Which cycle should be resolved first? This is solved by requiring sets of cycle resolutions to be "supported".

Definition 4.18. Let A be a set of cycle resolutions with $A' = \bigcup_{B \in A} B$, and I be an interpretation for program P. Then A is a *supported resolution set* of I if for all $B \in A$ and for all $L \in B$

- if **not** $L \notin B$, then **not** L is supported by $I \setminus A'$,
- otherwise L is supported as unknown by $I \setminus A'$.

Example 4.19 (Program 2). Suppose we have an interpretation {p, q, r, **not** p, **not** q, **not** r}. Then, both {{p, q}, {r}} and {{**not** p, **not** q}, {r}} are supported resolution sets. However, if we alter the rule p:- q. to be p:- q,r. then the second set above is not a supported resolution set.

Now we have enough tools define our function for computing models.

Definition 4.20. Let $\mathcal{S}$ = (PC,EC,L,G) be a semantics in resolution form, P be a program, and I be a set of interpretations for P. Then, $\mathbf{T}_{\mathbf{P}}^{\mathcal{S}}$ is a function mapping to and from sets of interpretations, and $I' \in \mathbf{T}_{\mathbf{P}}^{\mathcal{S}}(I)$ if and only if $\exists I'' \in I, \exists A \in \mathsf{PC}(I''), \exists B \in \mathsf{EC}(I'')$, such that $C \subseteq A \cup B$ is the largest supported resolution set of I'' and $I' = \mathbf{T}'_{\mathbf{p}}(I'' \setminus (\bigcup_{C' \in C} C'))$.

Theorem 4.21. *Let $\mathcal{S}$ =(PC,EC,L,G) be a semantics in resolution form, and P be a program. Let $\mathcal{F}$ be a function that maps from sets of interpretations to sets of interpretations such that for some set of interpretations I, $I' \in \mathcal{F}(I)$ if and only if $\exists I'' \in I . I' = (\mathbf{T}'_{\mathbf{p}})^{\infty}(\mathsf{L}(I''))$ and I' has no unresolved propositions.*
$\mathsf{G}(\mathcal{F}(hfp(\mathbf{T}_{\mathbf{P}}^{\mathcal{S}})))$ is the set of all models of P with respect to $\mathcal{S}$.

To prove theorem 4.21 we need to show three things. We must show that the herbrand fixed-point exists for $\mathbf{T}_{\mathbf{P}}^{\mathcal{S}}$, that $\mathsf{G}(\mathcal{F}(\mathrm{hfp}(\mathbf{T}_{\mathbf{P}}^{\mathcal{S}})))$ is always a set of models for program P(This also proves that for every resolution form there exists a semantics), and all semantics can be represented in resolution form with $\mathsf{G}(\mathcal{F}(\mathrm{hfp}(\mathbf{T}_{\mathbf{P}}^{\mathcal{S}})))$ as the set models for a program with respect to the semantics.

To show that the herbrand fixed-point exists for $\mathbf{T}_{\mathbf{P}}^{\mathcal{S}}$ we will will look at what is "known" about the propositions, and how that increases monotonically. We will represent this increase of information using *subsumption*. Informally, an interpretation subsumes another interpretation if it knows everything the second interpretation knows and possible more.

Definition 4.22. Let I_1 and I_2 be interpretations of some program P. We say $I_1 \sqsupseteq I_2$(I_1 subsumes I_2), if and only if

- $I_1 \subseteq I_2$, and
- $\forall L \in I_2, \textbf{not}\ L \notin I_2 \Rightarrow L \in I_1$.

Theorem 4.23. *The subsumes operator is transitive.*

Proof. Let I, J, K be interpretations such that $I \sqsupseteq J$ and $J \sqsupseteq K$. It is obvious that $I \subseteq K$, since subset is transitive. Also note that if $L \in K$ and **not** $L \notin K$ then $L \in J$ and **not** $L \notin J$(because it is a subset). Therefore $L \in I$. Thus, $I \sqsupseteq K$. □

Definition 4.24. Let A and B be sets of interpretations of some program P. We say $A \sqsupseteq B$(A subsumes B) if and only if $\forall I_1 \in A, \exists I_2 \in B. I_1 \sqsupseteq I_2$.

Lemma 4.25. *Let I be an interpretation for some program P and R be a set of positive and negative literals from a set of positive and even cycle resolutions selected for $\mathbf{T}'_{\mathbf{p}}(I)$. Then, if $\mathbf{T}'_{\mathbf{p}}(I) \sqsupseteq I$ then $\mathbf{T}'_{\mathbf{p}}(\mathbf{T}'_{\mathbf{p}}(I) \setminus R) \sqsupseteq \mathbf{T}'_{\mathbf{p}}(I)$.*

Proof. Assume the opposite. That is, $\mathbf{T}'_{\mathbf{p}}(I) \sqsupseteq I$, but it is not the case that $\mathbf{T}'_{\mathbf{p}}(\mathbf{T}'_{\mathbf{p}}(I) \setminus R) \sqsupseteq \mathbf{T}'_{\mathbf{p}}(I)$. There are two cases:

Case 1: $\mathbf{T}'_{\mathbf{p}}(\mathbf{T}'_{\mathbf{p}}(I) \setminus R) \not\sqsupseteq \mathbf{T}'_{\mathbf{p}}(I)$. Thus, $\exists L \in \mathbf{T}'_{\mathbf{p}}(\mathbf{T}'_{\mathbf{p}}(I) \setminus R)$ such that $L \notin \mathbf{T}'_{\mathbf{p}}(I)$. By definition, $\mathbf{T}'_{\mathbf{p}}$ cannot add those literals back. This means, there exists a rule in ext(P) with L as the head and every body literal in $\mathbf{T}'_{\mathbf{p}}(I)$ and not in R, but since $\mathbf{T}'_{\mathbf{p}}(I) \subseteq I$ we know that those body variables must have been in I, and thus L should have been in $\mathbf{T}'_{\mathbf{p}}(I)$. A contradiction.

Case 2: $\exists L \in \mathbf{T}'_{\mathbf{p}}(I)$ such that **not** $L \notin \mathbf{T}'_{\mathbf{p}}(I)$, but $L \notin \mathbf{T}'_{\mathbf{p}}(\mathbf{T}'_{\mathbf{p}}(I) \setminus R)$. Then for all rules in ext(P) with L as the head there exists at least one body literal that is not in $\mathbf{T}'_{\mathbf{p}}(I) \setminus R$. There are two subcases:

Subcase 1: $L \in R$. Then, L must be unresolved in $\mathbf{T}'_{\mathbf{p}}(I)$, and thus $L \in \mathbf{T}'_{\mathbf{p}}(I)$. A contradiction.

Subcase 2: $L \notin R$. The body literals cannot be in $\mathbf{T}'_{\mathbf{p}}(I)$, but there is at least one such body for which the body literals are in I because $L \in \mathbf{T}'_{\mathbf{p}}(I)$. Since **not** $L \notin \mathbf{T}'_{\mathbf{p}}(I)$ for all rules in ext(P) with **not** L as the head there exists at least one body literal not in I. This means the body literals used to place $L \in \mathbf{T}'_{\mathbf{p}}(I)$ must be resolved in I. Therefore they must be in $\mathbf{T}'_{\mathbf{p}}(I) \setminus R$, and thus $L \in \mathbf{T}'_{\mathbf{p}}(\mathbf{T}'_{\mathbf{p}}(I) \setminus R)$. A contradiction.

Thus, $\mathbf{T}'_{\mathbf{p}}(\mathbf{T}'_{\mathbf{p}}(I) \setminus R) \sqsupseteq \mathbf{T}'_{\mathbf{p}}(I)$. □

Lemma 4.26. *Let I be a set of interpretations for some program P. Then, for some semantics $\mathcal{S}$, $\mathbf{T}^{\mathcal{S}}_{\mathbf{P}}(\mathbf{T}^{\mathcal{S}}_{\mathbf{P}}(I)) \sqsupseteq \mathbf{T}^{\mathcal{S}}_{\mathbf{P}}(I)$ if $\mathbf{T}^{\mathcal{S}}_{\mathbf{P}}(I) \sqsupseteq I$*

Proof. $\forall I_1 \in \mathbf{T}^{\mathcal{S}}_{\mathbf{P}}(\mathbf{T}^{\mathcal{S}}_{\mathbf{P}}(I)), \exists I_2 \in \mathbf{T}^{\mathcal{S}}_{\mathbf{P}}(I)$ such that $I_1 = \mathbf{T}'_{\mathbf{p}}(I_2 \setminus R)$ where R is the set of positive and negative literals from the positive and even cycle resolutions selected for I_2, and there exists $I_3 \in I$ such that $I_2 = \mathbf{T}'_{\mathbf{p}}(I_3 \setminus B)$ and $I_2 \sqsupseteq I_3 \setminus B$, where B comes from the cycle resolutions for I_3. By lemma 4.25, since $I_1 \sqsupseteq I_2$ it must be the case $\mathbf{T}^{\mathcal{S}}_{\mathbf{P}}(\mathbf{T}^{\mathcal{S}}_{\mathbf{P}}(I)) \sqsupseteq \mathbf{T}^{\mathcal{S}}_{\mathbf{P}}(I)$. □

Using subsumption, we can use induction to show that hfp($\mathbf{T}^{\mathcal{S}}_{\mathbf{P}}$) exists.

Theorem 4.27. *Let P be a program, and $\mathcal{S}$ be a semantics in resolution form. Then hfp($\mathbf{T}^{\mathcal{S}}_{\mathbf{P}}$) exists.*

Proof. Note that if each step of the computation of hfp($\mathbf{T}^{\mathcal{S}}_{\mathbf{P}}$) subsumes the previous step then a fixed-point must be reached. This is because once a proposition is resolved it stays resolved and cannot change value.

It is easy to see that any interpretation subsumes lit (P). Therefore, $\mathbf{T}^{\mathcal{S}}_{\mathbf{P}}(\{\text{ lit }(P)\}) \sqsupseteq \{\text{ lit }(P)\}$ and by lemma 4.26 each step subsumes the previous. Thus the herbrand fixed-point of $\mathbf{T}^{\mathcal{S}}_{\mathbf{P}}$ exists. □

Now we must show that $\mathtt{G}(\mathcal{F}(\text{hfp}(\mathbf{T}_P^{\mathcal{S}})))$ is always a set of models for program P, and all semantics can be represented in resolution form with $\mathtt{G}(\mathcal{F}(\text{hfp}(\mathbf{T}_P^{\mathcal{S}})))$ as the set models for a program with respect to the semantics.

Lemma 3.21 provides three properties that imply that an interpretation is a model. So we only need to show that those properties hold for each member of $\mathtt{G}(\mathcal{F}(\text{hfp}(\mathbf{T}_P^{\mathcal{S}})))$ to show that they are all models.

Lemma 4.28. *Let $\mathcal{S} = (\mathtt{PC}, \mathtt{EC}, \mathtt{L}, \mathtt{G})$ be a semantics in resolution form, and P be a program. Then, $\mathtt{G}(\mathcal{F}(\mathit{hfp}(\mathbf{T}_P^{\mathcal{S}})))$ is a set of models for P.*

Proof. Since $\mathtt{G}(\mathcal{F}(\text{hfp}(\mathbf{T}_P^{\mathcal{S}}))) \subseteq \mathcal{F}(\text{hfp}(\mathbf{T}_P^{\mathcal{S}}))$, we only need to prove $\mathcal{F}(\text{hfp}(\mathbf{T}_P^{\mathcal{S}}))$ is a set of models. Let $M = \mathcal{F}(\text{hfp}(\mathbf{T}_P^{\mathcal{S}}))$. From lemma 3.21, all interpretations in M are models if we can show:

(1) there is no proposition that is unresolved in I,
(2) for all $L \in I$, L is supported by I with respect to P, and
(3) for all propositions p referenced by P such that p, **not** $p \notin I$, p is supported with unknown by I with respect to P.

It is apparent that for all $I \in M$, I satisfies property 1(because of $\mathcal{F}$). So, we must show that properties 2 and 3 hold for all $I \in M$.

Case 1(Property 2): It should be noted that for I to be in M, I must be a fixed-point of the $\mathbf{T}'_P$ operator. Thus for all $L \in I$ we know $L \in \mathbf{T}'_P(I)$, and by the definition of $\mathbf{T}'_P$, there must exist a rule in $\text{ext}(P)$ with L as the head and for each literal L' in the body of that rule, $L' \in I$. Therefore, L is supported by I w.r.t. P, and property 2 holds.

Case 2(Property 3): If there is no $\bot$ literals then, property 3 trivially holds. Therefore, assume there is at least one proposition such that it is $\bot$ in I. For all such propositions, p, we know that neither p nor **not** p are inductive. Otherwise by lemma 4.17 either p or **not** p would be in I. Thus, they must be coinductive. Let $I = \mathbf{T}'^{\infty}_P(\mathtt{L}(I'))$ for some interpretation I'. There are two possible ways p could become $\bot$ in I.

Case 1: Assume p was resolved by $\mathtt{L}$, and therefore is unresolved in I'. There must be a rule with p in the head and a rule with **not** p in the head such that the literals in both bodies are in I'. This comes directly from the definition of $\mathbf{T}'_P$ and the fact that I' is a fixed point. The only way a proposition can be unresolved is if its truth value depends on an odd cycle. The only way to resolve a proposition in an odd cycle is by making it unknown. Otherwise $\mathbf{T}'^{\infty}_P(\mathtt{L}(I')$ will never reach a fixed point.
Since, p is unresolved, we know that at least one literal in the body of each rule must be unresolved, and resolved the same way by $\mathtt{L}$. This means, that for those two rules, all body literals are either in $\mathtt{L}(I')$ or they are unknown in $\mathtt{L}(I')$, and therefore p must be supported with unknown by I.

Case 2: Assume p is $\bot$ in I'. Then, there must exist an interpretation I_2 generated while computing the Herbrand fixed-point of $\mathbf{T}_P^{\mathcal{S}}$ for which by repeatedly resolving cycles and applying $\mathbf{T}'_P$, I' will be generated, such that p is unresolved in I_2 but will be resolved in the next step. Since p is resolved to be unknown in the new interpretation, p must have been resolved by removing p and **not** p, and by definition p will be supported with unknown by the resulting interpretation. Since it is supported with unknown by that interpretation, we know that the value of p will not change due to $\mathbf{T}'_P$. Thus, p will be supported with unknown by I', and therefore I.

Therefore, property 3 holds, and all $I \in M$ are models of P.

□

Next we will show that all semantics can be represented in resolution form with $\mathtt{G}(\mathcal{F}(\text{hfp}(\mathbf{T}_\mathbf{P}^{\mathcal{S}})))$ as the set models for a program with respect to the semantics. This is more complicated so we will define some more tools to work with.

Lemma 4.29. *Let $\mathcal{S} = (f_p, f_e, f_o)$ be a semantics in proof model form, and P be a program. There exists a semantics in resolution form $\mathcal{S}' = (\mathtt{PC}, \mathtt{EC}, \mathtt{L}, \mathtt{G})$ such that $\mathtt{G}(\mathcal{F}(hfp(\mathbf{T}_\mathbf{P}^{\mathcal{S}'})))$ is the set of all models of P with respect to $\mathcal{S}$.*

To prove this we will construct $\mathcal{S}'$ and show that the resulting interpretations can be represented as proof models that are accepted by the proof model form. So, we need a way to convert between cycle resolutions and cycles. Each cycle set potentially represents two cycles. These cycles are the negation of each other. Resolving the cycle set by assigning true or false to the propositions involved will eliminate one of the cycles(it will be assigned false). When a cycle resolution resolves a cycle set by assigning $\perp$ to the propositions involved it does so to both cycles.

Definition 4.30. Let P be a program, C be a cycle set of P, L a literal and R a cycle resolution that resolves C with **not** $L \in R$. Then, $\text{cycle}(C, R, L) = L_1, L_2, \ldots, L_n$ for some $n > 0$ where

- $L_1 = L$ and $L_{n+1} = L$, and
- for all $0 < i \leq n$ if L_i is positive there exists a rule $r \in C$ such that $\text{head}(r) = L_i$ and $L_{i+1} \in \text{body}(r)$ otherwise there exists a rule $r \in C$ such that $\text{head}(r) = \text{prop}(L_i)$ and **not** $L_{i+1} \in \text{body}(r)$.

The truth value assigned to this cycle is $\perp$ if $L \in R$ and true otherwise.

To make this simpler, we will also split each function from the proof model form into three different functions depending on whether or not the whole model or set of models is needed to compute its result. This will allow us to separate the local and global information.

Definition 4.31. Let P be a program, I a proof model, and f be a function that maps a cycle, proof model, and program to a subset of $\{\text{true}, \perp\}$. We can define the functions $\mathtt{C}_f, \mathtt{L}_f, \mathtt{G}_f$ as follows.

- $\mathtt{C}_f(c) = f(c, I, P)$ if I and P are not used in the computation, and $\mathtt{C}_f(c) = \{\text{true}, \perp\}$ otherwise.
- $\mathtt{L}_f(c) = f(c, I, P)$ if P is not used in the computation but I is, and $\mathtt{L}_f(c) = \{\text{true}, \perp\}$ otherwise.
- $\mathtt{G}_f(c) = f(c, I, P)$ if P is used in the computation, and $\mathtt{G}_f(c) = \{\text{true}, \perp\}$ otherwise.

With these we can now prove lemma 4.29.

Proof of Lemma 4.29. We must construct PC, EC, L and G from $\mathcal{S}$. The function $\mathcal{R}_P^+$ and $\mathcal{R}_P^-$ will be functions that takes an interpretation and gives the set of all possible sets of cycle resolutions for C^+ and C^- respectively. That is,

$$\begin{aligned} \mathcal{R}_P^+(I) &= \{R \mid R \text{ resolves } C^+(I)\}, \text{ and} \\ \mathcal{R}_P^-(I) &= \{R \mid R \text{ resolves } C^-(I)\}. \end{aligned}$$

Let $F(X, Y)$ be a predicate that is true if and only if X is a set of cycle resolutions, Y is a function that maps a cycle to a subset of $\{\text{true}, \perp\}$, and for all $x \in X$, some $c \in C(I)$ and literal L such that **not** $L \in x$, the value assigned to $\text{cycle}(c, x, L)$ is in $Y(\text{cycle}(c, x, L))$. We will use the $\mathcal{R}$ functions to define PC and EC.

$$\begin{aligned} \mathtt{PC}(I) &= \{R \mid R \in \mathcal{R}_P^+(I), F(R, C_{f_p})\} \\ \mathtt{EC}(I) &= \{R \mid R \in \mathcal{R}_P^-(I), F(R, C_{f_e})\} \end{aligned}$$

L resolves odd cycles and makes use of interpretation level information to filter interpretations. It can do two things: resolve unresolved propositions or unresolve resolved propositions. Any

interpretation that has unresolved propositions after being filtered through L will be thrown away by the $\mathcal{F}$ function defined in theorem 4.21. So we will first try to resolve any unresolved propositions. We know that such propositions must depend on an odd cycle.

We first notice that if an odd cycle is assigned true we are assigning a proposition both true and false. So f_o must result in $\{\bot\}$ or $\{\}$. Let $\mathcal{I}(I) = (\mathbf{T}'_{\mathbf{P}})^{\infty}(I \setminus \{L \mid c \in C(I), c \text{ is odd}, R \subseteq I, \{L, \mathbf{not}\ L\} \subseteq R, R \text{ resolves } c, F(R, \mathsf{C}_{f_o}) \vee F(R, \mathsf{L}_{f_o})\})$.

For convenience we define the predicate F' where $F'_X(R)$ is true if and only if $F(R, X_{f_p}) \vee F(R, X_{f_e}) \vee F(R, X_{f_o})$. Then, $\mathsf{L}(I) = \mathcal{I}(I) \cup \{\mathbf{not}\ L \mid L \in \mathcal{I}(I), c \in C(\text{ lit }(P)), \neg \cdot R \subseteq \mathcal{I}(I), \mathbf{not}\ L \in R, R \text{ resolves } c, \neg F'_L(R)\}$.

Finally, we define G.

$$\mathsf{G}(M) = \{I \mid I \in M, \forall c \in C(\text{ lit }(P)) : \forall R \subseteq I : \neg \cdot R \text{ resolves } c \Rightarrow F'_G(\neg \cdot R)\}.$$

Now we must show that a model is in $\mathsf{G}(\mathcal{F}(\text{hfp}(\mathbf{T}_{\mathbf{P}}^{\mathcal{S}'})))$ if and only if the equivalent proof model is accepted by $\mathcal{S}$. Let $\mathcal{M}_P = \mathsf{G}(\mathcal{F}(\text{hfp}(\mathbf{T}_{\mathbf{P}}^{\mathcal{S}'})))$. If we assume that it is not the case, then either there is a model $M \in \mathcal{M}_P$ that is not accepted or there is a proof model that is accepted but its model is not in $\mathcal{M}_P$.

Case 1. Assume $M \in \mathcal{M}_P$ but proofmodel(M) is not accepted by $\mathcal{S}$. Then from definition 3.31, there exists $c \in$ proofmodel(M), with A being the set of all cycles for c, and all functions $\tau : A \to \{\text{true}, \bot\}$ such that $\bigwedge_{a \in A} (\tau(a) \in f(a, c, P) \wedge \tau(a)) \neq \text{label}(c)$. So, for all possible τ

(1) there exists $a \in A$ such that $\tau(a) \notin f(a, \text{proofmodel}(M), P)$,
(2) there exists $a \in A$ such that $\tau(a) = \bot$, $\bot \in f(a, \text{proofmodel}(M), P)$, and $\text{label}(c) = \text{true}$, or
(3) for all $a \in A$, $\tau(a) = \text{true}$, $\text{true} \in f(a, \text{proofmodel}(M), P)$, and $\text{label}(c) = \bot$.

We will construct a function τ_M such that $\tau_M(a)$ is the set containing only the truth value of L in I where L is a literal in a. Notice that all such literals must have the same value or M would be inconsistent with the program.

(1) Assume there exists $a \in A$ such that $\tau_M(a) \notin f(a, \text{proofmodel}(M), P)$. There exists some cycle set C, cycle resolution R, and literal L such that $\text{cycle}(C, R, L) = a$ with an associated value of $\tau_M(a)$ and R was used to resolve the propositions mentioned in a. However for R to be given by the cycle resolution functions $\tau_M(a) \in C_f$ for the corresponding cycle types function, f. Therefore since $\tau_M(a) \notin f(a, \text{proofmodel}(M), P)$ then either $\tau_M(a) \notin L_f(a)$ or $\tau_M(a) \notin G_f(a)$. But in the first case L would make the literals in a unresolved, and M would have been removed as a possible model, and in the second case G would remove M. Therefore, $\tau_M(a) \in f(a, \text{proofmodel}(M), P)$, which contradicts our assumption.
(2) Assume there exists $a \in A$ such that $\tau_M(a) = \bot$, $\bot \in f(a, \text{proofmodel}(M), P)$, and $\text{label}(c) = \text{true}$. But τ_M was defined such that $\tau_M(a) = \text{label}(c)$. This is clear to see if a is a direct cycle, and if a is an indirect cycle, it follows from lemma 3.29.
(3) Assume for all $a \in A$, $\tau_M(a) = \text{true}$, $\text{true} \in f(a, \text{proofmodel}(M), P)$, and $\text{label}(c) = \bot$. But τ_M was defined such that $\tau_M(a) = \text{label}(c)$ as explained in case 2.

Case 2. Assume a proof model M is accepted by $\mathcal{S}$, but model(M) $\notin \mathcal{M}_P$. There are four places model(M) could be eliminated when computing the models.

(1) model(M) could have been computed, but then removed by G,
(2) model(M) $\in \text{hfp}(\mathbf{T}_{\mathbf{P}}^{\mathcal{S}'})$ but a proposition in M becomes unresolved by L,
(3) there exists some $M' \in \text{hfp}(\mathbf{T}_{\mathbf{P}}^{\mathcal{S}'})$ that has an unresolved proposition that is not resolved by L, or

(4) there exists some step in the computation of the herbrand fixed point that contains an interpretation M' such that M' is subsumed by model(M) and there is no such interpretation in the next step.

To show a contradiction we must show that neither of these four possibilities applies to M.

Case 1. Since model(M) is removed by G, there exists a cycle set c and $R \subseteq$ model$M$ such that $\neg \cdot R$ resolves c but $F'_G(\neg \cdot R)$ is false. But $\neg \cdot R$ resolves c by assigning the involved propositions the same value M does. And since M is accepted by $\mathcal{S}$ that truth value must be in $G_{f_p}(c)$, $G_{f_e}(c)$, or $G_{f_o}(c)$. Which contradicts $F'_G(\neg \cdot R)$ being false.

Case 2. Since L unresolves a predicate in model(M), there exists a cycle set and cycle resolution R that resolves it such that $\neg \cdot R \subseteq$ model$M$ and $F(R, L_{f_p})$, $F(R, L_{f_e})$, and $F(R, L_{f_o})$ are all false. But R assigns the same value to the literals in the cycle as M does. Since M is accepted that truth value must be in the result of one of the L functions. Which contradicts the claim that the F predicate is false for R and the L functions.

Case 3. Since there is an unresolved literal when the interpretation is given to L we know that it must be part of an odd cycle. Since it remains unresolved afterwards either there is no cycle resolution to resolve the cycle set for that literal or $F(R, C_{ff_o})$ is false for all such cycle resolutions. We know that the first case cannot happen since M exists. But in the second case, there must be an R that makes F true since M is accepted by $\mathcal{S}$.

Case 4. Since M' is subsumed by model(M) there must a proposition, p that is unresolved in M', resolved in model(M), and cannot be resolved to be assigned the same truth value as it is in model(M). From lemmas 3.7 and 3.29 we know that the value of p must depend on a direct cycle. This means there is no cycle resolution in PC(M') or EC(M') which allows p to be assigned the same value as in model(M). But, for the cycle resolution, R, that assigns the same value as model(M) $F(R, C_{f_p})$ or $F(R, C_{f_e})$ must be true since M is accepted. Therefore, R must be given by PC or EC.

All cases lead to a contradiction therefore our claim must hold. □

Proof of Theorem 4.21. Proof follows from Lemma 4.28 and Lemma 4.29. □

Now we will take a closer look at using resolution form to define a semantics. From the proof for lemma 4.29, We can see that there is a limited number of ways to create resolutions for a cycle set. Since we depend only on the information contained in the cycle itself and there is no special propositions or metalogical features we only need to worry about the following.

Positive Cycles:
- The cycle contains no negations.
- The cycle contains all negations.

Even Cycles:
- There are two worlds, one where the cycle is assigned true, and one where it is assigned false.
- The cycle is assigned $\bot$.

Odd Cycles:
- The cycle is assigned $\bot$.
- The cycle cannot be resolved.

To see how cycle sets and resolutions are computed, consider program 2. In this program, we have two cycle sets: {p:− q, q:− p} and {r:− r}. So there are six positive cycle resolutions: {p, q}, {**not** p, **not** q}, {p, q, **not** p, **not** q}, {r}, {**not** r}, {r, **not** r}. Any of the first three resolutions with

one of the last three will form a complete set of resolutions for this program. It should also be noted that if we modified the first rule to become p:− q, s, the previous resolutions are still valid.

Next we define some useful functions that can be used to define the semantics presented in our background section. First presented are some positive cycle resolution functions. Since the only information about the positive cycles we can use is whether or not the cycle is negated, we can define the following: positive cycles are always false, positive cycles are always true, positive cycles are always ⊥, positive cycles create two worlds, one where it is true and one where it is false, and positive cycles create three worlds by assigning true, false, and ⊥ respectively.

We will start by defining a resolution function that resolves positive cycles by assigning false, like with well-founded and stable model semantics.

Definition 4.32. Let P be a program and I an interpretation of P. Let R be a set of positive cycle resolutions such that no resolution contains a negative literal, and R resolves $C^+(I)$. Then, let $\mathcal{P}_P^-$ be a positive cycle resolution function such that $\mathcal{P}_P^-(I) = \{R\}$

Example 4.33 (Program 2). If $I = \{\text{p}, \text{q}, \text{r}, \textbf{not}\ \text{p}, \textbf{not}\ \text{q}, \textbf{not}\ \text{r}\}$ then $\mathcal{P}_P^-(I) = \{\{\{\text{p}, \text{q}\}, \{\text{r}\}\}\}$.

As can be seen from the example, $\mathcal{P}_P^-$ does resolve all positive cycles by making them false. More formally:

Theorem 4.34. *For all programs P, proof models M of P, and positive cycles C for M, if a resolution form for a semantics uses $\mathcal{P}_P^-$, $f_p(C, M, P) = \{true\}$ if C has negative literals and $f(C, M, P) = \{\}$ otherwise.*

Proof. Let I be an interpretation of some program P such that there exists a model that subsumes I and for all $L \in I$ either L is unresolved or L is supported or supported as unknown by I. From theorem 4.9, for each positive cycle set C unresolved in I and resolution $R \in R'$ where $R' \in \mathcal{P}_P^-(I)$ there exists a cycle C' and coinductive proof c that assigns the same value to those literals. Notice that there is a model M that resolves that cycle set the same way such that $c \in \text{proofmodel}(M)$. Since R contains no negative literals it will assign false to the propositions, and the literals of C' will all be negative. In addition, there are no other resolutions to resolve C by definition of $\mathcal{P}_P^-$. So, for all cycles C' if the literals of C' are negative, $f_p(C', M, P) = \{\text{true}\}$, and $f_p(C', M, P) = \{\}$ otherwise. □

Now we want a resolution function that will assign true to positive cycles. None of our example semantics does this, but is included for completeness.

Definition 4.35. Let P be a program and I an interpretation of P. Let R be a set of positive cycle resolutions such that no resolution contains a positive literal, and R resolves $C^+(I)$. Then, let $\mathcal{P}_P^+$ be a positive cycle resolution function such that $\mathcal{P}_P^+(I) = \{R\}$

Example 4.36 (Program 2). If $I = \{\text{p}, \text{q}, \text{r}, \textbf{not}\ \text{p}, \textbf{not}\ \text{q}, \textbf{not}\ \text{r}\}$ then $\mathcal{P}_P^+(I) = \{\{\{\textbf{not}\ \text{p}, \textbf{not}\ \text{q}\}, \{\textbf{not}\ \text{r}\}\}\}$.

So, $\mathcal{P}_P^+$ resolves positive cycles by always making them true.

Theorem 4.37. *For all programs P, proof models M of P, and positive cycles C for M, if a resolution form for a semantics uses $\mathcal{P}_P^+$, $f_p(C, M, P) = \{true\}$ if C has no negative literals and $f(C, M, P) = \{\}$ otherwise.*

Proof. Let I be an interpretation of some program P such that there exists some model that subsumes I and for all $L \in I$ either L is unresolved or L is supported or supported as unknown by I. From theorem 4.9, For each positive cycle set C unresolved in I and resolution $R \in R'$

where $R' \in \mathcal{P}_P^+(I)$ there exists a cycle C' and coinductive proof c that assigns the same value to those literals. Notice that there is a model M that resolves that cycle set the same way such that $c \in \text{proofmodel}(M)$. Since R contains only negative literals it will assign true to the propositions, and the literals of C' will all be positive. In addition, there are no other resolutions to resolve C by definition of $\mathcal{P}_P^+$. So, for all cycles C' if the literals of C' are positive, $f_p(C', M, P) = \{\text{true}\}$, and $f_p(C', M, P) = \{\}$ otherwise. □

Next is a cycle resolution function that assigns $\bot$ to positive cycles. This would be used, for example, when defining Fitting's 3-value semantics.

Definition 4.38. Let P be a program and I an interpretation of P. Let R be a set of positive cycle resolutions such that $\forall r \in R \forall L \in r.\textbf{not}\ L \in r$, and R resolves $C^+(I)$. Then, let $\mathcal{P}_P^=$ be a positive cycle resolution function such that $\mathcal{P}_P^=(I) = \{R\}$

Example 4.39 (Program 2). If I = {p, q, r, **not** p, **not** q, **not** r} then $\mathcal{P}_P^=(I)$ = {{{p, **not** p, q, **not** q}, {r, **not** r}}}.

As intended $\mathcal{P}_P^=$ resolves positive cycles by always assigning $\bot$.

Theorem 4.40. *For all programs P, proof models M of P, and positive cycles C for M, if a resolution form for a semantics uses $\mathcal{P}_P^=$, $f_p(C, M, P) = \{\bot\}$.*

Proof. Let I be an interpretation of some program P such that there exists some model that subsumes I and for all $L \in I$ either L is unresolved or L is supported or supported as unknown by I. From theorem 4.9, For each positive cycle set C unresolved in I and resolution $R \in R'$ where $R' \in \mathcal{P}_P^=(I)$ there exists two cycles C_1', C_2' which are negations to each other and coinductive proofs c_1, c_2 that assigns the same value to those literals. Notice that there is a model M that resolves that cycle set the same way such that $c \in \text{proofmodel}(M)$. Since R contains both positive and negative literals for each proposition in the cycle it will assign $\bot$ to the propositions, and the literals of C_1', C_2' will be all positive and all negative, respectively. In addition, there are no other resolutions to resolve C by definition of $\mathcal{P}_P^=$. So, for all cycles C' $f_p(C', M, P) = \{\bot\}$. □

For semantics such as co-stable models, we need a cycle resolution function that uses multiple worlds to assign both true and false .

Definition 4.41. Let P be a program and I be an interpretation for P. Let R be the largest set of sets of positive cycle resolutions such that $\forall R' \in R, \forall r \in R', L \in r \Rightarrow \textbf{not}\ L \notin r$ and R' resolves $C^+(I)$. Then, let $\mathcal{P}_P^*$ be a positive cycle resolution function such that $\mathcal{P}_P^*(I) = R$.

Theorem 4.42. *For all programs P, proof models M of P, and positive cycles C for M, if a resolution form for a semantics uses $\mathcal{P}_P^*$, $f_p(C, M, P) = \{true\}$.Y*

Proof. $\mathcal{P}_P^*$ can be defined in terms of $\mathcal{P}_P^-$ and $\mathcal{P}_P^+$. Let I be an interpretation, $R_1 \in \mathcal{P}_P^-(I)$, and $R_2 \in \mathcal{P}_P^+(I)$. Then $R \in \mathcal{P}_P^*(I)$ if and only if R resolves $C^+(I)$ and $\forall r \in R$ either $r \in R_1$ or $r \in R_2$. So, a cycle could be assigned true if it is assigned true by either $\mathcal{P}_P^-$ or $\mathcal{P}_P^+$ and since one always assigns true for cycles with positive literals and the other always assigns true for cycles with negative literals it follows that $f_p(C, M, P) = \{\text{true}\}$. □

Finally, we give a cycle resolution function that uses multiple worlds to assign all three values to a positive cycle. This can be useful when specifying all models of a programs completion.

Definition 4.43. Let P be a program and I be an interpretation for P. Let R be the largest set of sets of positive cycle resolutions such that $\forall R' \in R$, R' resolves $C^+(I)$. Then, let $\mathcal{P}_P$ be a positive cycle resolution function such that $\mathcal{P}_P(I) = R$.

Theorem 4.44. *For all programs P, proof models M of P, and positive cycles C for M, if a resolution form for a semantics uses $\mathcal{P}_P^*$, $f_p(C, M, P) = \{true, \bot\}$.*

Proof. $\mathcal{P}_P$ can be defined in terms of $\mathcal{P}_P^*$ and $\mathcal{P}_P^=$. Let I be an interpretation. For all $R_1 \in \mathcal{P}_P^*(I)$, and $R_2 \in \mathcal{P}_P^=(I)$: $R \in \mathcal{P}_P(I)$ if and only if R resolves $C^+(I)$ and $\forall r \in R$ either $r \in R_1$ or $r \in R_2$. So, a cycle could be assigned a value to match either $\mathcal{P}_P^*$ or $\mathcal{P}_P^=$ and since one always assigns true and the other always assigns $\bot$ it follows that $f_p(C, M, P) = \{\text{true}, \bot\}$.

□

We have presented five ways to resolve positive cycle sets. There are two more ways(within the restrictions assumed in this paper) to resolve positive cycles by using multiple world to assign true /⊥ and false /⊥ . These ways seem less useful, and to save on space we will not include them in this paper. For even cycle sets, there are only two ways to form resolutions. We can assign $\bot$ as in well-founded semantics or create multiple worlds as in stable model semantics.

We will first give a cycle resolution function that assigns $\bot$ to an even cycle like well-founded semantics.

Definition 4.45. Let P be a program and I be an interpretation for P. Let R be a set of even cycle resolutions such that $\forall r \in R, \forall L \in r.$**not** $L \in r$, and R resolves $C^-(I)$. Then, let $\mathcal{E}_P^{WF}$ be an even cycle resolution function such that $\mathcal{E}_P^{WF}(I) = \{R\}$.

Example 4.46 (Program 1). Suppose we have an interpretation I = {p, q, r, s, **not** p, **not** q, **not** r, **not** s}. Then, $\mathcal{E}_P^{WF}(I) = \{\{\text{p, q, } \textbf{not}\text{ p, } \textbf{not}\text{ q}\}\}$.

As can be seen, $\mathcal{E}_P^{WF}$ does assign $\bot$.

Theorem 4.47. *For all programs P, proof models M of P, and even cycles C for M, if a resolution form for a semantics uses $\mathcal{E}_P^{WF}$, $f_p(C, M, P) = \{\bot\}$.*

Proof. Let I be an interpretation of some program P such that there exists some model that subsumes I and for all $L \in I$ either L is unresolved or L is supported or supported as unknown by I. From theorem 4.9, For each positive cycle set C unresolved in I and resolution $R \in R'$ where $R' \in \mathcal{E}_P^{WF}(I)$ there exists two cycles C_1', C_2' which are negations to each other and coinductive proofs c_1, c_2 that assigns the same value to those literals. Notice that there is a model M that resolves that cycle set the same way such that $c \in \text{proofmodel}(M)$. Since R contains both positive and negative literals for each proposition in the cycle it will assign $\bot$ to the propositions. In addition, there are no other resolutions to resolve C by definition of $\mathcal{E}_P^{WF}$. So, for all cycles C' $f_p(C', M, P) = \{\bot\}$. □

Another way to resolve an even cycle is to use multiple worlds like stable models.

Definition 4.48. Let P be a program and I be an interpretation for P. Let R be the largest set of sets of even cycle resolutions such that $\forall R' \in R, \forall r \in R', \forall L \in r.$**not** $L \notin r$, and $\forall R' \in R$, R' resolves $C^-(I)$. Then, let $\mathcal{E}_P^{SM}$ be an even cycle resolution function such that $\mathcal{E}_P^{SM}(I) = R$

Example 4.49 (Program 1). Suppose we have an interpretation I = {p, q, r, s, **not** p, **not** q, **not** r, **not** s}. Then, $\mathcal{E}_P^{SM}(I) = \{\{\text{p}, \textbf{not}\text{ q}\}, \{\text{q}, \textbf{not}\text{ p}\}\}$.

From the example we can see how $\mathcal{E}_P^{SM}$ uses multiple worlds to resolve an even cycle.

Theorem 4.50. *For all programs P, proof models M of P, and positive cycles C for M, if a resolution form for a semantics uses $\mathcal{E}_P^{SM}$, $f_p(C, M, P) = \{true\}$.*

Proof. Let I be an interpretation of some program P such that there exists a model that subsumes I and for all $L \in I$ either L is unresolved or L is supported or supported as unknown by I. From

theorem 4.9, For each positive cycle set C unresolved in I, $R' \in \mathcal{E}_P^{SM}(I)$, and resolution $R \in R'$ there exists a cycle C' and coinductive proof c that assigns the same value to those literals. Notice that there is a model M that resolves that cycle set the same way such that $c \in \text{proofmodel}(M)$. Since there is no way R can contain a literal and its negation there are only two possible resolutions for C, and R could be either. So, for all cycles C' $f_p(C', M, P) = \{\text{true}\}$. □

Furthermore, there is a cycle resolution function not described here that would use multiple worlds to assign $\bot$ to an even cycle or assign true and false to the literals as with $\mathcal{E}_P^{SM}$.

Below we provide two local filter functions. These functions only take into account odd cycles at the cycle level, and make no use of the interpretation level. We will also use the identity function as a global filter function. This is enough to define the semantics used in this paper, but theorem 4.21 does not make this assumption.

Definition 4.51. For an interpretation I, $\mathcal{L}^{WF}(I) = \{L : L \in I, \textbf{not } L \notin I\}$

$\mathcal{L}^{WF}$ filters interpretations by assigning all unresolved literals $\bot$. As stated earlier we assume all positive and even cycles will be resolved by the time the local filter function used. So, all unresolved propositions must depend on an odd cycle.

Theorem 4.52. *For all semantics that use $\mathcal{L}^{WF}$, $f_o(C, M, P) = \{\bot\}$ for all odd cycles C, proof models M, and programs P such that M is a proof model of P and C is a cycle of M.*

Proof. By the definition of the resolution form of a semantics and the fact we have reached a fixed-point of $\mathbf{T}_\mathbf{P}^S$ it can be seen that when $\mathcal{L}^{WF}$ is applied to an interpretation all unresolved literals must depend on an odd cycle. For such literals, $\mathcal{L}^{WF}$ removes them. This has the same result as applying a cycle resolution to each odd cycle set that is comprised of the head of the rules in the odd cycle and their negations, and then applying $\mathbf{T}'_\mathbf{p}$ until we reach a fixed-point. From theorem 4.9, for each such odd cycle set C there are two proof cycles C'_1, C'_2 for some proof models M_1, M_2 of program P where C'_2 can be generated by negating each literal in C'_1 and both are assigned $\bot$. Therefore, $f_o(C'_1, M_1, P) = \{\bot\}$ and $f_o(C'_2, M_2, P) = \{\bot\}$. In addition, there are no other resolutions, and therefore the theorem holds. □

$\mathcal{L}^{SM}$ is an identity function for interpretation. Since the only unresolved literals in an interpretation given to $\mathcal{L}^{SM}$ should be part of an odd cycle, we can just keep them and from the definition of $\mathbf{T}_\mathbf{P}^S$ it will be eliminated as a possible model. This is for semantics such a stable models that cannot have odd cycles.

Definition 4.53. For an interpretation I, $\mathcal{L}^{SM}(I) = I$.

Theorem 4.54. *For all semantics that use $\mathcal{L}^{SM}$, odd cycles C, proof models M, and programs P, $f_o(C, M, P) = \{\}$.*

Proof. By the definition of the resolution form of a semantics and the fact we have reached a fixed-point of $\mathbf{T}_\mathbf{P}^S$ it can be seen that when $\mathcal{L}^{SM}$ is applied to an interpretation all unresolved literals must depend on an odd cycle. Since $\mathcal{L}^{SM}$ makes no changes to the interpretation, any unresolved literals stay unresolved, and the interpretation will be removed from the final set of interpretations. Assigning any truth value to an odd proof cycle contradicts this. Therefore, for all C, M, and P it must be the case that $f_o(C, M, P) = \{\}$. □

Finally, we will define the global filter function we will use for the rest of this paper.

Definition 4.55. For a set of interpretations I, $\mathcal{G}(I) = I$.

Using different combinations of the above cycle resolution and filter functions we can define any of the semantics presenting in section 2.4.

5 PROOF-THEORETIC FORMALIZATION

5.1 3-value Modified CoSLD Resolution

Since we will be working with 3-value logics such as well founded semantics we must modify the algorithm from [13] further. To do this we must differentiate between the truth value of a proposition and the success/failure of its proof. We will say that a query succeeds if there exists a model such that the query is not false in that model.

Definition 5.1. 3-value Modified CoSLD resolution can be defined by modifying the original algorithm as follows:

- The CHS is the call stack. A separate Partial Candidate Model (PCM) is used to record the model during execution.
- On success, the literals on the stack are assigned a value in reverse order.
- On coinductive success, the last literal on the call stack is not assigned a value.
- If a literal is to be assigned $\bot$, it is assigned the value temporarily and execution continues to the next branch. If a success assigns true to the literal the previous $\bot$ value is overwritten and true is assigned to the literal. Otherwise it stays $\bot$.

5.2 Restrictions

Besides the obvious restrictions that the semantics must use negation-as-failure and be a completion semantics, we impose some additional restrictions for the proof-theoretic method.

- All semantics that require a filter function besides the three defined in section 4 are unsupported. It is important to note that this is not a technical restriction, but one of convenience. All such semantics can be implemented by computing the consistency constraint imposed by the filter function and appending it to the query as we do for $\mathcal{L}^{SM}$.
- We will assume that no semantics will allow a cycle to be resolved as both true/false and $\bot$. This restriction can be lifted by non-deterministically selecting a resolution rule and trying again if needed.

5.3 Preprocessing

The goal directed algorithm presented in this paper is a generalization of the algorithm for stable models semantics presented in [13] and demonstrated in [14]. More details on preprocessing a program can be found in those papers.

5.3.1 Internal IDs. The method we will describe will require modifying the original program internally. This includes the generation of the consistency check as well as the creation of the extended program. This will sometimes require the use of new propositions. We want to hide these new propositions so that when the algorithm is viewed as a black box the modification is not apparent. So, we will need a means of marking these propositions. For the purpose of this paper we will surround a normal proposition name with "⟨" and "⟩" to represent an *internal name*. It is important to note that sample and ⟨*sample*⟩ are considered different propositions.

5.3.2 Dual Rule Generation. The method for generating the dual rules to be added to the extended program presented in section 2.1 of this paper is not suitable for practical applications. For the proof-theoretic algorithm we will use the method presented in [13].

To generate the extended program we add rules as follows:

Definition 5.2. Let P be some program. Then, for all propositions $\mathrm{p} \in \mathrm{props}(P)$:

- Collect all rules $r \in P$ for which $\mathrm{head}(r) = \mathrm{p}$.

- If no such rules exist add the rule "**not** p." otherwise:
  - for each rule r collected and each literal $L \in \text{body}(r)$, add a rule "$\langle not\_p_r \rangle$:– **not** L.", and
  - add a rule r' with head(r') = **not** p and the conjunction of all $\langle not\_p_r \rangle$ generated in the above rule.

5.3.3 Consistency Check. The consistency check is a rule for which its head is added to any queries to enforce the local filter function, and only $\mathcal{L}^{SM}$ requires a global constraint. If we assume all positive and even cycles are resolved before reaching the filter function, then the only unresolved propositions that can be present in an interpretation are those that are dependent on an odd cycle. This is consistent with fixed-point form and [13], and the same consistency check(also called an NMR check) can be used.

To generate the check we must first construct a call graph for the program, and decide what rules in the program form odd cycles. Each rule that is part of an odd cycle is called an OLON(Odd loop over negation) rule.

Definition 5.3. Let P be some program.

- For each OLON rule r :h:–$L_1, L_2, \ldots, L_n$ create a new proposition $\langle chk\_h_r \rangle$ and for each literal L_i, such that $L_i \neq$ **not** h, add a new rule "$\langle chk\_h_r \rangle$:– **not**L_i". Then, add rule "$\langle chk\_h_r \rangle$:– h".
- Create a new rule, r', with head(r') =$\langle chk \rangle$ and the conjunction of all $\langle chk\_h_r \rangle$'s from the previous step as the body.

5.4 The Rules

A specific semantics is specified by three rules. Each rule decides how to resolve a cycle when detected.

Definition 5.4. A *cycle resolution rule* can be one of three possible rules:

- SUCCESS(True) means a goal that results in a cycle will succeed with intended value true.
- SUCCESS($\bot$) means a goal that results in a cycle will succeed with the intended value $\bot$.
- FAIL means a goal that results in a cycle will fail.

In addition to the above rules, odd cycle resolutions rules must also specify whether or not a consistency check is needed. This will be represented in this paper as CHK and NOCHK.

Definition 5.5. A cycle resolution rule can be *fixed* or *symmetric*. A fixed cycle resolution rule applies to both the positive and negative goals. A symmetric cycle resolution rule will invert the truth value for negative goals. All rules are assumed to be symmetric unless specified with FIX. FAIL and SUCCESS(True) are symmetric of each other and SUCCESS($\bot$) is symmetric of itself.

We will assume that if a FAIL is FIXed there will be some sort of consistency check to ensure that the model does not have any cycles of that type. For our work we will only allow FAIL to be FIXed for odd cycles for which we already have a consistency check. With the current restrictions there is no way to determine if an even cycle should fail or if its negation should fail. So, we will also require even cycle rules to be FIXed.

Semantics	Positive Cycles	Even Cycles	Odd Cycles
Fittings 3-Val	SUCCESS(⊥)	SUCCESS(⊥) FIX	SUCCESS(⊥) NOCHK
Well-Founded	FAIL	SUCCESS(⊥) FIX	SUCCESS(⊥) NOCHK
Stable-Models	FAIL	SUCCESS(True) FIX	FAIL FIX,CHK
CoStable Models	SUCCESS(True) FIX	SUCCESS(True) FIX	FAIL,FIX,CHK

5.5 The Algorithm

The following algorithm assumes that the program had already been transformed with dual rules and the consistency check, and that cycle resolutions rules for positive, even, and odd cycles have been defined. We present the algorithm in a top-down manner, with the mutually recursive functions `prove_goals` and `prove_goal` as the core. Given some list of goals Q, `query(`Q`)` computes the partial model, for which each member of Q is not false , if it exists and fails otherwise.

```
query([L_1, L_2, ..., L_n]) begin
    if CHK then
        Let (T,PCM) ← prove_goals([L_1, L_2, ..., L_n, ⟨chk⟩],[],{})
    else
        Let (T,PCM) ← prove_goals([L_1, L_2, ..., L_n],[],{})
    end
    if T = False then
        FAIL
    else
        SUCCESS with PCM as the partial model
    end
end
```

`prove_goals` tries to find a proof for a conjunction of goals while constructing the partial candidate model.

The `prove_cycle` function is the coinductive portion of the algorithm. It searches the call stack to see if the current goal(or its negation) is already in it, signaling a cycle. If the current proof depends on a cycle, `prove_cycle` also detects what kind of cycle it is and applies the proper rule to resolve it.

The `apply_*_cycle_rule` functions above represent assigning a truth value based on the rule for the cycle. False is used to represent `FAIL`. If the argument to the function is negative and the rule is not `FIX`ed then the symmetric value is used.

Next, `prove_goal` tries to find a proof for a single goal by expanding rules.

When computing a model, the proof-theoretic algorithm is essentially computing the inductive and coinductive proofs. Any literal needed to prove the query or affected by the consistency check will have a proof computed for it. Any literals not in the resulting partial model can be computed independently and added to the ones computed for the partial model to form a proof model that is accepted by the semantics. It is important to note that the proof model does exist since odd cycles are the only way to invalidate a potential model (with the current restrictions).

```
prove_goals(Goals,CallStack,PCM) begin
    Let [L_1, L_2, ..., L_n] for some n ≥ 0 be a permutation of Goals if n = 0 then
        return (True, PCM)
    else
        for x ∈ PCM do
            if x = (L_1, T) then
                return (T, PCM)
            end
            if x = (not L_1, ⊥) then
                return (⊥, PCM)
            end
            if x = (not L_1, True) then
                return (False, PCM)
            end
        end
        Let T = prove_cycle(L_1, CallStack) if T ≠ NOCYCLE then
            return (T, PCM)
        else
            Let (T, PCM2) = prove_goal(L_1, CallStack, PCM) if T = False then
                return (T, {}, {})
            else
                Let (T2, PCM2) ← prove_goals([L_2, ..., L_n], CallStack, PCM2) if T2 = True
                 then
                    return (T, PCM2)
                else
                    return (T2, PCM2)
                end
            end
        end
    end
end
```

To prove our claim, we must have a way to convert to and from proof model form.

Definition 5.6. Let $\mathcal{R}$ be a cycle resolution rule. The *inverse* of $\mathcal{R}$ is

$$\neg\mathcal{R} = \begin{cases} \texttt{FAIL} & \text{if } \mathcal{R} = \texttt{SUCCESS(True)} \\ \texttt{SUCCESS(True)} & \text{if } \mathcal{R} = \texttt{FAIL} \\ \texttt{SUCCESS}(\bot) & \text{if } \mathcal{R} = \texttt{SUCCESS}(\bot) \end{cases}$$

Definition 5.7. D Let $\mathcal{R}$ be a cycle resolution rule.

$$\text{truthset}\,(\mathcal{R}) = \begin{cases} \{T\} & \text{if } \mathcal{R} = \texttt{SUCCESS}(T), \text{ where } T \in \{\text{true}\,, \bot\,\} \\ \{\} & \text{if } \mathcal{R} = \texttt{FAIL} \end{cases}$$

Definition 5.8. Let $\mathcal{S}$ be a semantics represented as cycle resolution rules. Let $\mathcal{R}_p, \mathcal{R}_e, \mathcal{R}_o$ be the cycle resolution rules for positive, even, and odd cycles, respectively. Then to_pmf($\mathcal{S}$) = (f_p, f_e, f_o) is a proof model form, and defined as follows.

- For some program P, proof model M and positive cycle C,
  - if C contains no negative literals then $f_p(C, M, P) =$ truthset $(\mathcal{R}_p)$,

```
prove_cycle(L,CallStack) begin
    Let CS ← CallStack Let NegCycle ← False while CS ≠ [] do
        Let CS = [L′ | CS2] if L′ is positive and L is negative then
            Let NegCycle ← True
        else if L′ is negative and L is positive then
            Let NegCycle ← True
        end
        if L′ = L then
            if NegCycle then
                Let X ← apply_even_cycle_rule(L)
            else
                Let X ← apply_positive_cycle_rule(L)
            end
            return X
        else if L′ = not L then
            Let X ← apply_odd_cycle_rule(L) return X
        end
        Let CS ← CS2
    end
    return NOCYCLE
end

prove_goal(L,CallStack,PCM) begin
    Let RS be a list of the bodies of all rules with L as the head Let Unknown ← False while
     RS ≠ [] do
        Let RS = [R | RS2] Let (T,PCM2) ← prove_goals(R,[L|CallStack],PCM) if
         T = True then
            if L is an internal id then
                return (True, PCM2)
            else
                return (True, PCM2 ∪ {(L, True)})
            end
        else if T = ⊥ then
            Let Unknown ← True Let PCM ← PCM2
        end
    end
    if Unknown then
        if L is an internal id then
            return (⊥, PCM)
        else
            return (⊥, PCM ∪ {(L, ⊥)})
        end
    else
        return (False, PCM)
    end
end
```

  - if C contains negative literals and $\mathcal{R}_p$ is FIXed then $f_p(C, M, P) =$ truthset $(\mathcal{R}_p)$, and
  - if C contains negative literals and $\mathcal{R}_p$ is not FIXed then $f_p(C, M, P) =$ truthset $(\neg\mathcal{R})$.
- For some program P, proof model M, and even cycle C, $f_e(C, M, P) =$ truthset $(\mathcal{R}_e)$.
- For some program P, proof model M, and odd cycle C,
  - if $\mathcal{R}_o$ requires a CHK then $f_o(C, M, P) = \{\}$, otherwise
  - $f_o(C, M, P) =$ truthset $(\mathcal{R})$.

Our algorithm generates partial models, which have a different format from interpretations as presented in this paper. So we will present some tools for working with them.

Definition 5.9. Let M_1 and M_2 be partial models. M_1 *conflicts with* M_2 if there exists a pair $(L_1, T_1) \in M_1$ and pair $(L_2, T_2) \in M_2$ such that

- $L_1 =$ **not** L_2 and either $T_1 \neq \bot$ or $T_2 \neq \bot$, or
- $L_1 = L_2$ and $T_1 \neq T_2$.

L_1 and L_2 are called *conflicting* literals, and we say L_1 *conflicts with* M_2 and L_2 *conflicts with* M_1.

Definition 5.10. Let S be the semantics represented by cycle resolution rules. Let Q be a list of literals to be proved and P be a program such that query(Q) succeeds with partial model M. Then, we can construct a coinductive proof set, proofset(M), from the execution. For each literal that succeeds coinductively, we can construct a coinductive proof with that literal as the root. The label is the value assigned on success, and the children will be the coinductive proofs of the coinductive literals in the body of the rule used to prove it. At the point where the coinductive success is determined we can use the previously constructed tree forming an infinite tree.

To prove the correctness of our algorithm we will show that the query can be extended until a full model is generated. We will show that that model is a superset of the original partial model and that it is a model with respect to the semantics. We will be using our conversion between cycle resolution rules and proof model form to show that.

Lemma 5.11. *Let $\mathcal{S}$ be a semantics represented by cycle resolution rules. Let P be a program with at least one model with respect to $\mathcal{S}$, M a partial model of P, and L a coinductive literal of P that succeeds. If for all partial models, D, that is generated when L succeeds(ignoring any consistency check) there exists some proposition p that is assigned different values by D and M then **not** L can succeed and there exists a partial model D' generated when **not** L succeeds(ignoring any consistency check) such that D' does not conflict M.*

Proof. If D conflicts with M then either the conflicting literal in M is the negation of the one in D(and one is not assigned $\bot$) or they are the same, but one assigns $\bot$ and the other assigns true. Since $\bot$ can only be assigned by a cycle resolution function, there must be a D that assigns the same as M which violates the assumption that that cannot happen. So the second case cannot happen, and we only need to consider the first case.

Since p does not have to be prop(L) we will have to consider the distance(number of rules) between L and p. We will ignore internal names when computing distance. In addition, we will assume that all literals between L and p are not assigned a value by D. If such a literal was assigned a value by D and it was conflicting we could use it instead of p, and if it was not conflicting then there is a way to make it succeed without a conflict, eliminating the need for p.

There can be multiple ways of proving a literal using different rules. We will induct on the maximum distance. Out of all possible ways of proving a literal(conforming with our assumption above) we will choose the one with the highest distance.

Base Case. There is zero maximum distance. This means L directly conflicts with M. But, then there is a way for its negation to succeed since it is in M.

Inductive Hypothesis. Assume if there is a maximum distance of k or less, **not** L succeeds and there exists a partial model D' generated when **not** L succeeds such that $D' \subseteq M$.

Inductive Step. Assume there is a distance of $k+1$. L can be positive or negative.

If L is positive then for each rule there exists a literal L' in the body that when it succeeds some proposition p is assigned a value that conflicts with M, and there is a k or less distance between L' and p. By the inductive hypothesis, the negation of each L' can succeed with a partial model that is a subset of M. By the definition of dual rules there is a way using those literals to make **not** L succeed with a partial model that is made by taking the union of the partial models for each L' and adding a truth assignment for **not** L. Such a partial model cannot conflict with M.

If L is negative, by construction there are one or more internal literals in the body. One such literal has one or more rules, each with a single literal in the body. Since L always conflicts with M it must be the case that each such rule that can succeed leads to a conflict. So each body literal must have a maximum distance of at most k. By the inductive hypothesis, the negation of those literals can succeed without conflicting with M. By the construction of dual rules, those literals correspond to the body literals of one rule that has **not** L as the head. So **not** L can succeed with a partial model comprised of the union of all the partial models generated by the body literals and an assignment for **not** L. This partial model cannot conflict with M.

Therefore, by induction we conclude that **not** L can succeed with a partial model that does not conflict M. □

Definition 5.12. Let c_1, c_2 be coinductive proofs. Then c_1 *depends* on c_2 if and only if

- $c_2 \in \text{support}(c_1)$ or
- there exists $c_3 \in \text{support}(c_1)$ such that c_3 depends on c_2.

If there does not exist c' such that c_1 depends on c' and c' depends on c_2 then c_2 is said to be *most depended.* depends(c_1) is the set of all coinductive proofs c_2 such that c_1 depends on c_2.

Lemma 5.13. *Let S be the semantics represented by cycle resolution rules. Let Q be a list of literals to be proved and P be a program such that* query(Q) *succeeds with partial model M. For all coinductive $p \in$ props(P), either* query([$p|Q$]) *or* query([***not*** $p|Q$]) *succeeds with martial model M' such that proofset (M') covers proofset (M).*

Proof. First, it must be shown that either p or **not** p must succeed. So, let L be a literal such that prop(L) = p. Since L is coinductive, we know that the value of L depends on some cycle. If that cycle should FAIL then unless that cycle is odd then the cycles negation should succeed with SUCCESS(true), and **not** L is in that cycle. Now we must show that in the case the cycle should FAIL the cycle cannot be odd. Since we require odd cycle resolution rules that FAIL to be FIXed and require a CHK, it must be the case that query(Q) must have proved the consistency check. By definition, if the consistency check succeeds then for each OLON rule, r, either head(r) can succeed or some $L \in$ body(r) allow **not** head(r) to succeed. Since that query succeeded it must be the case that a FAIL happens because of an odd cycle in P.

Now we have three possibilities: p fails, **not** p fails, both p and **not** p succeed. The first two cases are symmetrical. So we can combine them.

- Let L be a literal such that prop(L) = p. Assume **not** L fails. Then, query([**not** $L|Q$]) must fail. We must show that query([$L|Q$]) can succeed with a partial model M' such that proofset (M') covers proofset (M). Since L can succeed, if the query fails then every partial model generated when L succeeds must conflict with M. But, by lemma 5.11 **not** L must

succeed, contradicting our assumption that it fails. Therefore query([L|Q]) must succeed, and there must be a partial model generated when L succeeds that does not conflict with M. Since M could be generated again when Q and any consistency check succeeds, the resulting partial model M' will be the union of the two. Clearly this partial model is a superset of M, and proofset(M') covers proofset(M).

- Suppose both p and **not** p can succeed. Let L be a literal such that prop(L) = p. Assume that query([**not** L|Q]) fails. Since **not** L can succeed, it must be the case that for every partial model D that can be generated when L succeeds D conflicts with M. By lemma 5.11, L must be able to succeed with a partial model D' such that D' does not conflict with M. This means there is a partial model $D' \cup M$ that can be generated when query([L|Q]) succeeds. This partial model is clearly a superset of M, and thus proofset($D' \cup M$) covers proofset(M).

□

Theorem 5.14. *Let Q be a list of literals to be proved, and P be a program. Let S be the semantics represented by cycle resolution rules. Then,*

(1) query(Q) *succeeds with partial model M implies the literals in Q are in M and there exists a model M' of P with respect to S such that $M \subseteq M'$, and*
(2) *if there exists a model M' of P with respect to S with the literals of Q are in M' there exists $M \subseteq M'$ with the literals of Q in M such that* query(Q) *succeeds with partial model M.*

Proof. **Claim 1.** Assume query(Q) succeeds with partial model M. Then, we can construct a coinductive proof set, C, from the execution. For each literal that succeeds coinductively, we can construct a coinductive proof with that literal as the root. The label is the value assigned on success, and the children will be the coinductive proofs of the coinductive literals in the body of the rule used to prove it. At the point where the coinductive success is determined we can use the previously constructed tree forming an infinite tree.

We must show that there exists some proof model that covers C and is accepted by to_pmf($\mathcal{S}$). To show that such a proof model exists we must show that C is consistent and that we can consistently make it complete. Suppose, C is not consistent. That means either there is a literal that is true in one coinductive proof but $\bot$ in another, or there is a literal that is true in one coninductive proof but its negation is either true or $\bot$ in another. In both cases, prove_goals ensures that this does not happen by checking the PCM. This way, all contradictions will fail.

Now we must show that there is a proof model for P that covers C. It is trivial to see that C covers itself. Let $X \subseteq$ lit (P) be the largest set of coinductive literals such that the literal and its negation do not have a coinductive proof in C and prepending Q with the literals in X the resulting query will succeed with partial model M. proofmodel(M) must be complete, consistent, and valid. It is complete since there must be a coinductive proof for each coinductive proposition or its negation by lemma 5.13. We can take each coinductive proposition and by 5.13 by prepending the proposition or its negation the resulting query will succeed. Therefore the proposition or its negation must be in X. It must be consistent or the query would have failed when prove_goals checks the PCM. It must be valid since prove_goal will assign true if there exists a rule that evaluates to true, and assign $\bot$ only if a rule evaluates to $\bot$ and no rule evaluates to true . This is part of the $RS \neq []$ loop.

Therefore, we must be able to extend C to a proof model. Now, we must show that there exists such a model that is accepted by to_pmf($\mathcal{S}$). Suppose proofmodel(M) is not accepted. Then, from definition 3.31, there exists $c \in$ proofmodel(M), with B being the set of all cycles

for c, and all functions $\tau : B \rightarrow \{\text{true}, \bot\}$ such that $\bigwedge_{a \in B} (\tau(a) \in f(a, \text{proofmodel}(M), P) \wedge \tau(a)) \neq \text{label}(c)$. So, for all possible τ

(1) there exists $a \in B$ such that $\tau(a) \notin f(a, \text{proofmodel}(M), P)$,
(2) there exists $a \in B$ such that $\tau(a) = \bot$, $\bot \in f(a, \text{proofmodel}(M), P)$, and $\text{label}(c) = \text{true}$, or
(3) for all $a \in B$, $\tau(a) = \text{true}$, $\text{true} \in f(a, \text{proofmodel}(M), P)$, and $\text{label}(c) = \bot$.

Let τ_C be defined such that τ_C maps a cycle to the associated label from c.

(1) Suppose there exists $a \in B$ such that $\tau_C(a) \notin f(a, \text{proofmodel}(M), P)$. We know that the label of c is determined by the cycle resolution rule. Therefore $\tau_C(a) \in f(a, \text{proofmodel}(M), P)$ by the definition of to_pmf($\mathcal{S}$) which contradicts our assumption.
(2) There cannot exist $a \in B$ such that $\tau(a) = \bot$, $\bot \in f(a, \text{proofmodel}(M), P)$, and $\text{label}(c) = \text{true}$ since the value of $\tau_C(a)$ is the same as the labels of the coinductive proofs whose literals form a. Otherwise, c would not meet the definition of a coinductive proof.
(3) It cannot be the case that for all $a \in B$, $\tau(a) = \text{true}$, $\text{true} \in f(a, \text{proofmodel}(M), P)$, and $\text{label}(c) = \bot$ since the value of $\tau_C(a)$ is the same as the labels of the coinductive proofs whose literals form a. Otherwise, c would not meet the definition of a coinductive proof.

Therefore proofmodel(M) is accepted by to_pmf($\mathcal{S}$).

Claim 2. Assume there is a model M' with respect to S such that all literals in Q are in M'. We must show that query(Q) succeeds with some partial model M, the literals in Q are in M, and $M \subseteq M'$. Assume the opposite is true. That is either query(Q) fails, there is a literal in Q that is not in M or $M \nsubseteq M'$.

Case 1. Assume query(Q) fails. There is some literal, L, in the consistency check or in Q for which prove_cycle or prove_goal returns False in the first call to prove_goals. If a consistency check for a rule in an odd cycle fails, this means all literals that do not depend on the odd cycle will succeed. So, if L is in the consistency check, then there must be an odd cycle. Since, there is a consistency check, then f_o for the semantics will be false for all odd cycles, and therefore any proof models(and therefore models) that contain an odd cycle will not be accepted by S. Thus, L cannot be in the consistency check.

Base Case. Assume for some lists of literals Q' with all members in M', S, and partial candidate model M_2, prove_goals(Q', S, M_2) returns False directly without recursive calls. There are two possibilities. Either prove_cycle or prove_goal returns False .

If prove_cycle returns False , then the current stack and the first literal in Q' form a cycle that uses a FAIL rule. Therefore, for that cycle the corresponding proof model form predicate will always be false. This contradicts the assumption that it is in M'.

prove_goal can only return False directly if there is no rules with L in the head. This contradicts the assumption that $L \in M'$.

Inductive Hypothesis. Suppose for a list of literals Q' such that all literals in Q' are in M' it is a contradiction that prove_goals(Q', [],PCM) returns False in k or less recursive calls.

Inductive Step. Suppose prove_goals returns False after $k + 1$ recursive calls. Using the same logic as in the base case, we know that prove_cycle cannot

return False in this case. So it must be `prove_goal` that returned False . The only way for this to happen is if every recursive call to `prove_goals` return False . But each call will return False within k recursive calls, which by the inductive hypothesis is a contradiction.

By induction, L cannot be in Q, and thus `query`(Q) failing will always lead to a contradiction.

Case 2. Assume `query`(Q) succeeds with a PCM, M, but there exists a literal in Q that is not in M. This, obviously cannot happen since that literal has to succeed when calling `prove_goal`, and it will place the literal into the PCM when returning.

Case 3. Assume `query`(Q) succeeds with a PCM, M, but all such M are not a subset of M'. There must be a literal in M that is not in M', but only those required to prove Q and any consistency check will be in M. Since M' also contains all the literals required to prove Q then there must be a partial model containing only those. A contradiction. □

6 RELATED WORK

In his papers, [3] and [4], Jürgen Dix explores properties of semantics for normal logic programs. The semantics looked at in his papers include the semantics considered in this paper while also considering many others. Dix's work does not try to generalize semantics as we do. Instead it looks at how different semantics are similar and dissimilar and how the various properties enable or restrict the use of a semantics.

More recently, Scott D. Stoller and Yanhong A. Liu has also done work in unifying the semantics discussed in this paper, but have taken a different approach. They designed two new semantics(founded semantics and constraint semantics) that subsume the other semantics. Instead of a parameterized algorithm for computing models, their semantics make use of metalogical properties that are assigned to predicates to determine how they are handled. This allows them to simulate the behavior of the other semantics, and can even simulate other semantics not covered by our work.[12]

It is our belief, however, that with some modifications to our algorithm and assumptions, such as allowing metalogical properties and modifying $\mathbf{T}'_{\mathrm{p}}$ to use dual rules only for *complete*(a metalogical property) predicates instead of all predicates, we can compute models for the semantics in [12].

7 CONCLUSION

In this paper we demonstrated that normal logic program semantics, for which the models of a program is a subset of its completion, make use of a combination of induction and coinduction. We explored the role of both induction and coinduction, and showed that the major difference between such semantics is in how they assign values to cyclic dependent computations. We then presented the declarative and operational semantics of various semantics of normal logic programs in a unifying, systematic manner; considering four semantics for normal logic programs (Fitting???s 3-valued semantics, well-founded semantics, stable model semantics, and co-stable model semantics) and how they relate to our approach.

In section 3 we presented a formalization of the role of induction and coinduction. This formalization also served to bridge the gap between the model theoretic and proof theoretic approaches by assigning literals in a model a proof in addition to a truth value. We showed how, within some reasonable restrictions, we can represent all semantics in terms of how cycles are handled in these proofs.

Section 4 presented a fixed-point declarative semantics, and proved its equivalence with the previous formalization. This fixed-point formalization constructs the set of all models for a program by starting from the set of all its positive and negative literals (representing having no information about the model) and removing literals (ignoring literals that form an odd cycle) in each iteration when we know that cannot be true. Multiple worlds such as those generated by even cycles in stable model semantics are represented by creating two models in the next step. One will have the proposition removed and the other will have its negation removed. Finally when a fixed point is reached, odd cycles are resolved before any remaining models that do not conform to the current semantics based on all cycles or even other models are removed.

Finally, we gave a parametric goal-directed algorithm for computing partial models of these semantics. The psudocode for the algorithm, example executions, and proof of correctness can be found in section 5.

REFERENCES

[1] Krzysztof R. Apt and Roland N. Bol. 1994. Logic Programming and Negation: A Survey. *J. Log. Program.* 19/20 (1994), 9–71. DOI: http://dx.doi.org/10.1016/0743-1066(94)90024-8

[2] Keith Clark. 1978. Negation as Failure. In *Logic and Data Bases*, H. Gallaire and J. Minker (Eds.). Plenum, New York, 293–322.

[3] Jürgen Dix. 1995. A Classification Theory of Semantics of Normal Logic Programs: I. Strong Properties. (1995).

[4] Jürgen Dix. 1995. A Classification Theory of Semantics of Normal Logic Programs: II. Weak Properties. (1995).

[5] Melvin Fitting and Marion Ben-Jacob. 1988. Stratified and Three-valued Logic Programming Semantics. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*. 1054–1069.

[6] Michael Gelfond and Vladimir Lifschitz. 1988. The Stable Model Semantics for Logic Programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*. 1070–1080.

[7] Gopal Gupta, Kyle Marple, Brian DeVries, Feliks Kluniak, Richard Min, Neda Saeedloei, and Tobias Haage. 2012. Coinductive Answer Set Programming or Consistency-based Computing. (2012). https://contraintes.inria.fr/~saeedloe/program.html Co-LP 2012 - A workshop on Coinductive Logic Programming.

[8] Gopal Gupta, Neda Saeedloei, Brian DeVries, Richard Min, Kyle Marple, and Feliks Kluźniak. 2011. Infinite Computation, Co-induction and Computational Logic. In *Algebra and Coalgebra in Computer Science*, Andrea Corradini, Bartek Klin, and Corina Cîrstea (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 40–54.

[9] Gopal Gupta, Elmer Salazar, Kyle Marple, Zhuo Chen, and Farhad Shakerin. 2017. A Case for Query-driven Predicate Answer Set Programming. In *ARCADE 2017. 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements (EPiC Series in Computing)*, Giles Reger and Dmitriy Traytel (Eds.), Vol. 51. EasyChair, 64–68. DOI: http://dx.doi.org/10.29007/ngm2

[10] Bart Jacobs and Jan Rutten. 1997. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin* 62 (1997), 62–222.

[11] Tom Leinster. 2016. Basic Category Theory. Cambridge Studies in Advanced Mathematics, Vol. 143, Cambridge University Press, 2014, (2016).

[12] Yanhong A. Liu and Scott D. Stoller. 2016. The Founded Semantics and Constraint Semantics of Logic Rules. *CoRR* abs/1606.06269 (2016). http://arxiv.org/abs/1606.06269

[13] Kyle Marple, Ajay Bansal, Richard Min, and Gopal Gupta. 2012. Goal-directed execution of answer set programs. In *Principles and Practice of Declarative Programming, PPDP'12, Leuven, Belgium - September 19 - 21, 2012*. 35–44. DOI: http://dx.doi.org/10.1145/2370776.2370782

[14] Kyle Marple and Gopal Gupta. 2012. Galliwasp: A Goal-Directed Answer Set Solver. In *Logic-Based Program Synthesis and Transformation, 22nd International Symposium, LOPSTR 2012, Leuven, Belgium, September 18-20, 2012, Revised Selected Papers*. 122–136. DOI: http://dx.doi.org/10.1007/978-3-642-38197-3_9

[15] Kyle Marple, Elmer Salazar, and Gopal Gupta. 2017. Computing Stable Models of Normal Logic Programs Without Grounding. *arXiv preprint arXiv:1709.00501* (2017).

[16] Richard Kyunglib Min. 2009. *Predicate Answer Set Programming with Coinduction.* Ph.D. Dissertation. Richardson, TX, USA. Advisor(s) Gupta, Gopal. AAI3375960.

[17] J. Minker (Ed.). 1988. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers Inc. 19–88 pages.

[18] Grigore Roşu and Dorel Lucanu. 2009. Circular Coinduction: A Proof Theoretical Foundation. In *Algebra and Coalgebra in Computer Science*, Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 127–144.

[19] Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. 2007. Co-Logic Programming: Extending Logic Programming with Coinduction. In *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*. 472–483.

[20] Allan Van Gelder, Kennith A. Ross, and John S. Schlipf. 1991. The well-founded semantics for general logic programs. *Journal of the Association for Computing Machinery* 38, 03 (July 1991), 620–650.

APPENDIX

A RESOLUTION FORM FOR VARIOUS SEMANTICS

We will specifically take a look at well founded semantics, stable models semantics and costable models semantics.

We will define well-founded semantics first. As stated in section 2.4, well-founded semantics resolves positive cycles by assigning false to the propositions, and both even and odd cycles are resolved by assigning $\bot$. So, we use the corresponding cycle resolution functions to define that behavior.

THEOREM A.1. *$(\mathcal{P}_P^-, \mathcal{E}_P^{WF}, \mathcal{L}^{WF}, \mathcal{G})$ is the resolution form of well-founded semantics.*

PROOF. We can construct a proof model form by using the equivalent functions.

- $f_p(C, M, P) = \{\text{true}\}$ if C contains negations and $f_m(C, M, P) = \{\}$ otherwise,
- $f_e(C, M, P) = \{\bot\}$, and
- $f_o(C, M, P) = \{\bot\}$.

Now we want to show that an interpretation I is a well-founded semantics model of some program P if an only if its proof model is accepted.

Suppose I is a well founded model of P, but proofmodel(I) is not accepted by f_p, f_e, f_o. Then from definition 3.31, there exists $c \in$ proofmodel(I), with A being the set of all cycles for c, and all functions $\tau : A \longrightarrow \{\text{true}, \bot\}$ such that $\bigwedge_{a \in A} (\tau(a) \in f(a, c, P) \wedge \tau(a)) \neq \text{label}(c)$. So, for all possible τ

(1) there exists $a \in A$ such that $\tau(a) \notin f(a, C, P)$,
(2) there exists $a \in A$ such that $\tau(a) = \bot$, $\bot \in f(a, C, P)$, and $\text{label}(c) = \text{true}$, or
(3) for all $a \in A$, $\tau(a) = \text{true}$, $\text{true} \in f(a, C, P)$, and $\text{label}(c) = \bot$.

Now, we will construct a function τ_I such that for each cycle in c we assign the associated value for that cycle. From our assumption, one of the three possibilities hold.

Case 1. There exists $a \in A$ such that $\tau_I(a) \notin f(a, C, P)$. We know $\tau_I(A) \neq$ false since the value came from c and can only be true or $\bot$.

- Suppose a is a positive cycle. Then the propositions that form a form an unfounded set, and will therefore be assigned false in I. This means a must be a negated positive cycle, and it must have an associated value of true in c. But then $\tau_I(a) \in f(a, C, P)$.
- Suppose a is not a positive cycle. We know that $f(a, C, P) = \{\bot\}$. So it must be the case that $\tau_I(a) = \text{true}$. Since, c assigned true to the cycle this means the literals must be true in I. That is the literals in a must be in I. Since I is a well founded model, if such a literal is positive then T_P must have added the literal to it and if the literal is negative it must have been in an unfounded set. If T_P added the literal, then we can trace the rules needed to add it and the rules to add its body literals, and so on, and construct either an inductive or coinductive proof. The only way to construct a coinductive proof is if some of the rules formed a positive cycle. Since the literal is positive it cannot be part of that cycle. Otherwise it would be unfounded and therefore be false in I. So, if

the literal cannot be part of any cycle; contradicting that the literal is in a. Therefore, the literal must be negative, and its proposition is part of an unfounded set. There are two ways to be part of an unfounded set. For each rule with the proposition as the head, the rule leads to a positive cycle or some body literal is false in the previous step of computing the model. In the second case, if the body literal is negative, then its proposition must have been added by T_P, but we already established that would mean the literal is not part of a cycle, which contradicts that the literal is in a. Therefore we only need to consider the literal being part of a positive cycle, but that contradicts our assumption that a is not a positive cycle.

Therefore, it must be the case $\tau_I(a) \in f(a, C, P)$.

Case 2. There exists $a \in A$ such that $\tau_I(a) = \bot$, $\bot \in f(a, C, P)$, and $\text{label}(c) = \text{true}$. Since, $\tau_I(a) = \bot$, we know that a was assigned $\bot$ by c, and therefore by the definition of coinductive proofs, $\text{label}(c) = \bot$. This contradicts our assumption.

Case 3. For all $a \in A$, $\tau(a) = \text{true}$, $\text{true} \in f(a, C, P)$, and $\text{label}(c) = \bot$. Similarly to case 2, by the definition of coinductive proofs $\text{label}(c)$ must be true. Another contradiction.

Therefore, we know that τ_I does not satisfy any of the three possibilities, which contradicts the assumption that all such functions must. Therefore proofmodel(I) must be accepted by f_p, f_e, f_o.

Lastly, we must show the opposite. That is, if proofmodel(I) is accepted by f_p, f_e, f_o then I must be a well-founded model. From theorem 3.7 we know that all inductive literals must be in the model. From lemma 3.29 we know the value of coinductive literals depends entirely on direct cycles. So, we only need to consider literals that are part of direct cycles. T_P cannot compute the values of cycles. So, the only way for a literal to be placed in the model is through the unfounded set.

The propositions of the literals in a positive cycle always forms an unfounded set, and the negation of the literals will be placed in the model. For even and odd cycles, they can never be in an unfounded set. These cycles are comprised of both positive and negative literals(even if you negate the cycle). This means there is at least one literal that is negative, and therefore cannot be the head of a rule, and since we are dealing with a direct cycle of a coinductive proof, we know there is no other way for that literal to be false in the model without the proposition being inductive. Therefore, the literals of odd and even cycles cannot be in the model, and therefore are assigned the value of $\bot$.

Notice that since proofmodel(I) is accepted by f_p, f_e, f_o the values assigned to the literals of direct cycles must match the required assignments above, and proofmodel(I) must be a proof model. Since proofmodel is a proof model I must be a model, and since literals in the cycles are assigned values consistent with well-founded semantics, I must be a well-founded model. □

Next we define stable model semantics. Stable model semantics resolves positive cycles in the same way as well-founded semantics: by assigning false to the propositions in the cycle. It uses multiple worlds, allowing a proposition to be true in one world and false in another, to resolve even cycles. Finally, any odd cycle will lead to an inconsistency, not allowing such assignments to be models. Once again, we chose the cycle resolution functions previously defined in this section that corresponds to that behavior.

Theorem A.2. *$(\mathcal{P}_P^-, \mathcal{E}_P^{SM}, \mathcal{L}^{SM}, \mathcal{G})$ is the resolution form of stable models semantics.*

Proof. We can construct a proof model form by using the equivalent functions.

- $f_p(C, M, P) = \{\text{true}\}$ if C contains negations and $f_p(C, M, P) = \{\}$ otherwise,
- $f_e(C, M, P) = \{\text{true}\}$, and
- $f_o(C, M, P) = \{\}$.

Now we want to show that an interpretation I is a stable models semantics model of some program P if an only if its proof model is accepted.

Suppose I is a stable model of P, but proofmodel(I) is not accepted by f_p, f_e, f_o. Then from definition 3.31, there exists $c \in$ proofmodel(I), with A being the set of all cycles for c, and all functions $\tau : A \rightarrow \{\text{true}, \bot\}$ such that $\bigwedge_{a \in A} (\tau(a) \in f(a, c, P) \wedge \tau(a)) \neq \text{label}(c)$. So, for all possible τ

(1) there exists $a \in A$ such that $\tau(a) \notin f(a, C, P)$,
(2) there exists $a \in A$ such that $\tau(a) = \bot$, $\bot \in f(a, C, P)$, and $\text{label}(c) = \text{true}$, or
(3) for all $a \in A$, $\tau(a) = \text{true}$, $\text{true} \in f(a, C, P)$, and $\text{label}(c) = \bot$.

Now, we will construct a function τ_I such that for each cycle in c we assign the associated value for that cycle. From our assumption, one of the three possibilities hold.

Case 1. There exists $a \in A$ such that $\tau_I(a) \notin f(a, C, P)$. We know $\tau_I(A) \neq$ false since the value came from c and can only be true or $\bot$.

- Suppose a is a positive cycle. Since the cycle exists the rules that produce that cycle must be in the reduct. But, since the rules are cyclic they will not be in the least model. Since I is a stable model, the propositions that form a will be assigned false in I. This means a must be a negated positive cycle, and it must have an associated value of true in c. But then $\tau_I(a) \in f(a, C, P)$.
- Suppose a is an even cycle. Since a is a cycle for c it must have an associated value of true or $\bot$. Since stable models semantics is two value we know that $\tau_I(a) \neq \bot$, and therefore it must be the case that $\tau_I(a) = \text{true}$. But, $\text{true} \in f(a, C, P)$.
- If true is assigned to a literal in an odd cycle that would mean both that literal and its negation will be assigned true. This is obviously impossible, and since stable models is two values it cannot be assigned $\bot$. Thus, a cannot be an odd cycle.

Therefore, it must be the case $\tau_I(a) \in f(a, C, P)$.

Case 2. There exists $a \in A$ such that $\tau_I(a) = \bot$, $\bot \in f(a, C, P)$, and $\text{label}(c) = \text{true}$. Since, I is a stable model which is two-value, it can never be the case that $\tau_I(a) = \bot$.

Case 3. For all $a \in A$, $\tau(a) = \text{true}$, $\text{true} \in f(a, C, P)$, and $\text{label}(c) = \bot$. Similarly to case 2, it will never be the case $\text{label}(c) = \bot$.

Therefore, we know that τ_I does not satisfy any of the three possibilities, which contradicts the assumption that all such functions must. Therefore proofmodel(I) must be accepted by f_p, f_e, f_o.

Lastly, we must show the opposite. That is, if proofmodel(I) is accepted by f_p, f_e, f_o then I must be a stable model. So, let M be the least model of the residual program with respect to I. Now, assume proofmodel(I) is accepted by f_p, f_e, f_o but I is not a stable model. There exists a literal L such that $L \in M \iff L \notin I$. From theorem 3.7 we know that all inductive literals must be in the model. From lemma 3.29 we know the value of coinductive literals depends entirely on direct cycles. So, we only need to consider literals that are part of direct cycles.

Case 1. Assume $L \in I$.

- Suppose L is part of a positive cycle for proofmodel(I). Since $L \in I$ we know that it must be assigned true in proofmodel(I), and because proofmodel(I) is accepted L must be a negated literal. Since the cycle exists there must be rules that produce the cycle with the head of one of the rules being prop(L). These cycles cannot have any negated literals in the body that are false in I or else the cycle wouldn't be produced. Therefore, they produce cyclic rules in the reduct, and prop(L) cannot be in M, and therefore $L \in M$. A contradiction.
- Suppose L is part of an even cycle for proofmodel(I). From lemma 3.29 we can assume the even cycle is what causes L to not be in M. Since $L \in I$ we know that it must be

assigned true in proofmodel(I). If L is positive, then we know that any rules that have **not** L in the body will not be in the reduct. Since L negatively depends on the heads of those rules(by definition of an even cycle) then L must be in M. If, on the other hand, L is negative then we know there is some positive L' in the even cycle. Any rules that have **not** L' in the body will not be in the reduct. So either **not** L will have no rules, or any rules for **not** L will have a proposition in the body that has no rules. Therefore L must be in M. A contradiction.

- Since proofmodel(I) is accepted there can be no odd cycles.

Case 2. Assume $L \in M$.

- Suppose **not** L is part of a positive cycle for proofmodel(I). Since **not** $L \in I$ we know that it must be assigned true in proofmodel(I), and because proofmodel(I) is accepted L must be a positive literal. Since the cycle exists there must be rules that produce the cycle with the head of one of the rules being L. Since $L \in M$ there must be another rule with L as the head that puts $L \in M$. This will be inductive, and therefore the original rule must depend on some rule with some negative literals in the body and is part of a cycle set. Those literals must be in I or the rule would have been removed. But, that extra rule will be accounted for and must be false for the dual rule to be used to construct a coinductive proof. A contradiction.
- Suppose **not** L is part of an even cycle for proofmodel(I). From lemma 3.29 we can assume the even cycle is what causes L to be in M. Since **not** $L \in I$ we know that it must be assigned true in proofmodel(I). If L is negative, then we know that any rules that have L in the body will not be in the reduct. Since **not** L negatively depends on the heads of those rules(by definition of an even cycle) then **not** L must be in M and thus $L \notin M$. If, on the other hand, L is positive then we know there is some negative L' in the even cycle. Any rules that have L' in the body will not be in the reduct. So either L will have no rules, or any rules for L will have a proposition in the body that has no rules. Therefore **not** L must be in M. Meaning $L \notin M$. A contradiction.
- Since proofmodel(I) is accepted there can be no odd cycles.

Thus, by contradiction, I must be a stable model.

□

Finally, here is the definition of co-stable model semantics. Co-stable model semantics treats both even and odd cycles the same way as stable models. So we will use the same cycle resolution functions for these types of cycles. However, positive cycles are handled using multiple worlds. The propositions in a positive cycle are assigned true in one world and false in another.

Theorem A.3. *$(\mathcal{P}_P, \mathcal{E}_P^{SM}, \mathcal{L}^{SM}, \mathcal{G})$ is the resolution form of co-stable models semantics.*

Proof. We can construct a proof model form by using the equivalent functions.

- $f_p(C, M, P) = \{\text{true}\}$,
- $f_e(C, M, P) = \{\text{true}\}$, and
- $f_o(C, M, P) = \{\}$.

Now we want to show that an interpretation I is a co-stable models semantics model of some program P if an only if its proof model is accepted.

Suppose I is a stable model of P, but proofmodel(I) is not accepted by f_p, f_e, f_o. Then from definition 3.31, there exists $c \in$ proofmodel(I), with A being the set of all cycles for c, and all functions $\tau : A \to \{\text{true}, \bot\}$ such that $\bigwedge_{a \in A} (\tau(a) \in f(a, c, P) \wedge \tau(a)) \neq \text{label}(c)$. So, for all possible τ

(1) there exists $a \in A$ such that $\tau(a) \notin f(a, C, P)$,

(2) there exists $a \in A$ such that $\tau(a) = \bot$, $\bot \in f(a, C, P)$, and $\text{label}(c) = \text{true}$, or
(3) for all $a \in A$, $\tau(a) = \text{true}$, $\text{true} \in f(a, C, P)$, and $\text{label}(c) = \bot$.

Now, we will construct a function τ_I such that for each cycle in c we assign the associated value for that cycle. From our assumption, one of the three possibilities hold.

Case 1. There exists $a \in A$ such that $\tau_I(a) \notin f(a, C, P)$. We know $\tau_I(A) \neq \text{false}$ since the value came from c and can only be true or $\bot$.
- Suppose a is a positive or an even cycle. Since a is a cycle for c it must have an associated value of true or $\bot$. Since co-stable models semantics is two value we know that $\tau_I(a) \neq \bot$, and therefore it must be the case that $\tau_I(a) = \text{true}$. But, $\text{true} \in f(a, C, P)$.
- If true is assigned to a literal in an odd cycle that would mean both that literal and its negation will be assigned true. This is obviously impossible, and since stable models is two values it cannot be assigned $\bot$. Thus, a cannot be an odd cycle.

Therefore, it must be the case $\tau_I(a) \in f(a, C, P)$.

Case 2. There exists $a \in A$ such that $\tau_I(a) = \bot$, $\bot \in f(a, C, P)$, and $\text{label}(c) = \text{true}$. Since, I is a co-stable model which is two-value, it can never be the case that $\tau_I(a) = \bot$.

Case 3. For all $a \in A$, $\tau(a) = \text{true}$, $\text{true} \in f(a, C, P)$, and $\text{label}(c) = \bot$. Similarly to case 2, it will never be the case $\text{label}(c) = \bot$.

Therefore, we know that τ_I does not satisfy any of the three possibilities, which contradicts the assumption that all such functions must. Therefore proofmodel(I) must be accepted by f_p, f_e, f_o.

Lastly, we must show the opposite. That is, if proofmodel(I) is accepted by f_p, f_e, f_o then I must be a costable model. So, let M be the least model of the residual program with respect to I. Now, assume proofmodel(I) is accepted by f_p, f_e, f_o but I is not a stable model. There exists a literal L such that $L \in M \iff L \notin I$. From theorem 3.7 we know that all inductive literals must be in the model. From lemma 3.29 we know the value of coinductive literals depends entirely on direct cycles. So, we only need to consider literals that are part of direct cycles.

Case 1. Assume $L \in I$.
- Suppose L is part of a positive or even cycle for proofmodel(I). Since $L \in I$ we know that it must be assigned true in proofmodel(I). In addition, there exists a rule for L that was used to compute the coinductive proof in proofmodel(I) and so all body literals of that rule must be in I. If L is positive, the fact for L will be in the coreduct, and therefore $L \in M$. If L is negative, then by the definition of dual rules, we know that for all rules with **not** L as the head there exists some body literal that is not in I. Therefore, there will be no rules for **not** L in the coreduct, and $L \in M$.
- Since proofmodel(I) is accepted there can be no odd cycles.

Case 2. Assume $L \in M$.
- Suppose **not** L is part of a positive or even cycle for proofmodel(I). Since **not** $L \in I$ we know that it must be assigned true in proofmodel(I). In addition, there exists a rule for **not** L that was used to compute the coinductive proof in proofmodel(I) and so all body literals of that rule must be in I. If L is negative, the fact for **not** L will be in the coreduct, and therefore **not** $L \in M$ and $L \notin M$. If L is positive, then by the definition of dual rules, we know that for all rules with L as the head there exists some body literal that is not in I. Therefore, there will be no rules for L in the coreduct, and $L \notin M$.
- Since proofmodel(I) is accepted there can be no odd cycles.

Thus, by contradiction, I must be a co-stable model.

□

B EXAMPLES FOR GOAL-DIRECTED ALGORITHM

We will now present some examples of executing this algorithm. We will take a look at our three basic example programs; representing the three cycle types. For each program the result of preprocessing the program will be given before the execution of the algorithm, and two executions will be given using different semantics to highlight how the algorithm behaves for the different cycles. For these examples we will select rules from top to bottom, and goals from left to right.

For our first example consider program 2. We will be using Well founded and coStable models semantics to illustrate how the algorithm behaves on positive cycles.

Example B.1. Program 2

p :- q.
q :- p.
r :- r.
%Dual Rules
$\langle not\_p_0 \rangle$:- **not** q.
not p :- $\langle not\_p_0 \rangle$.

$\langle not\_q_0 \rangle$:- **not** p.
not q :- $\langle not\_q_0 \rangle$.

$\langle not\_r_0 \rangle$:- **not** r.
not r :- $\langle not\_r_0 \rangle$.

$\langle chk \rangle$.

Well-founded:

```
⊣ query([⟨not_q⟩, r])
  ⊣ prove_goals([⟨not_q⟩, r], [], {})
    ⊣ prove_cycle(⟨not_q⟩, []) = NOCYCLE
    ⊣ prove_goal(⟨not_q⟩, [], {})
      ⊣ prove_goals([⟨not_q0⟩], [⟨not_q⟩], {})
        ⊣ prove_cycle(⟨not_q0⟩, [⟨not_q⟩]) = NOCYCLE
        ⊣ prove_goal(⟨not_q0⟩, [⟨not_q⟩], {})
          ⊣ prove_goals([⟨not_p⟩], [⟨not_q0⟩, ⟨not_q⟩], {})
            ⊣ prove_cycle(⟨not_p⟩, [⟨not_q0⟩, ⟨not_q⟩]) = NOCYCLE
            ⊣ prove_goal(⟨not_p⟩, [⟨not_q0⟩, ⟨not_q⟩], {})
              ⊣ prove_goals([⟨not_p0⟩], [⟨not_p⟩, ⟨not_q0⟩, ⟨not_q⟩], {})
                ⊣ prove_cycle(⟨not_p0⟩, [⟨not_p⟩, ⟨not_q0⟩, ⟨not_q⟩]) = NOCYCLE
                ⊣ prove_goal(⟨not_p0⟩, [⟨not_p⟩, ⟨not_q0⟩, ⟨not_q⟩], {})
                  ⊣ prove_goals([⟨not_q⟩], [⟨not_p0⟩, ⟨not_p⟩, ⟨not_q0⟩, ⟨not_q⟩], {})
                    ⊣ prove_cycle(⟨not_q⟩, [⟨not_p0⟩, ⟨not_p⟩, ⟨not_q0⟩, ⟨not_q⟩])
                      ⊣ NegCycle = False
                      ⊣ apply_positive_cycle_rule(⟨not_q⟩) = True
                      ⊣ return True
                    ⊣ return (True, {})
                  ⊣ return (True, {(⟨not_p0⟩,True)})
                ⊣ prove_goals([], [⟨not_p⟩, ⟨not_q0⟩, ⟨not_q⟩], {(⟨not_p0⟩,True)})
```

```
                    ⊣ return (True, {(⟨not_p0⟩,True)})
                  ⊣ return (True, {(⟨not_p0⟩,True)})
                ⊣ return (True, {(⟨not_p⟩,True), (⟨not_p0⟩,True)})
              ⊣ prove_goals([], [⟨not_q0⟩, ⟨not_q⟩], {(⟨not_p⟩,True), (⟨not_p0⟩,True)})
                ⊣ return (True, {(⟨not_p⟩,True), (⟨not_p0⟩,True)})
              ⊣ return (True, {(⟨not_p⟩,True), (⟨not_p0⟩,True)})
            ⊣ return (True, {(⟨not_q0⟩,True), (⟨not_p⟩,True), (⟨not_p0⟩,True)})
          ⊣ prove_goals([], [⟨not_q⟩], {(⟨not_q0⟩,True), (⟨not_p⟩,True),
                                                                  (⟨not_p0⟩,True)})
            ⊣ return (True, {(⟨not_q0⟩,True), (⟨not_p⟩,True), (⟨not_p0⟩,True)})
          ⊣ return (True, {(⟨not_q0⟩,True), (⟨not_p⟩,True), (⟨not_p0⟩,True)})
        ⊣ return (True, {(⟨not_q⟩,True), (⟨not_q0⟩,True), (⟨not_p⟩,True),
                                                                  (⟨not_p0⟩,True)})
      ⊣ prove_goals([r], [], {(⟨not_q⟩,True), (⟨not_q0⟩,True), (⟨not_p⟩,True),
                                                                  (⟨not_p0⟩,True)})
        ⊣ prove_cycle(r, []) = NOCYCLE
        ⊣ prove_goal(r, [], {(⟨not_q⟩,True), (⟨not_q0⟩,True), (⟨not_p⟩,True),
                                                                  (⟨not_p0⟩,True)})
          ⊣ prove_goals([r], [r], {(⟨not_q⟩,True), (⟨not_q0⟩,True), (⟨not_p⟩,True),
                                                                  (⟨not_p0⟩,True)})
            ⊣ prove_cycle(r, [r])
              ⊣ NegCycle = False
              ⊣ apply_positive_cycle_rule(r) = False
              ⊣ return False
            ⊣ return (False, {(⟨not_q⟩,True), (⟨not_q0⟩,True), (⟨not_p⟩,True),
                                                                  (⟨not_p0⟩,True)})
          ⊣ return (False, {(⟨not_q⟩,True), (⟨not_q0⟩,True), (⟨not_p⟩,True),
                                                                  (⟨not_p0⟩,True)})
        ⊣ return (False, {})
      ⊣ return (False, {})
    ⊣ FAIL
```

coStable Models:

```
⊣ query([⟨not_q⟩, r])
  ⊣ prove_goals([⟨not_q⟩, r, ⟨chk⟩], [], {})
    ⊣ prove_cycle(⟨not_q⟩, []) = NOCYCLE
    ⊣ prove_goal(⟨not_q⟩, [], {})
      ⊣ prove_goals([⟨not_q0⟩], [⟨not_q⟩], {})
        ⊣ prove_cycle(⟨not_q0⟩, [⟨not_q⟩]) = NOCYCLE
        ⊣ prove_goal(⟨not_q0⟩, [⟨not_q⟩], {})
          ⊣ prove_goals([⟨not_p⟩], [⟨not_q0⟩, ⟨not_q⟩], {})
            ⊣ prove_cycle(⟨not_p⟩, [⟨not_q0⟩, ⟨not_q⟩]) = NOCYCLE
            ⊣ prove_goal(⟨not_p⟩, [⟨not_q0⟩, ⟨not_q⟩], {})
              ⊣ prove_goals([⟨not_p0⟩], [⟨not_p⟩, ⟨not_q0⟩, ⟨not_q⟩], {})
                ⊣ prove_cycle(⟨not_p0⟩, [⟨not_p⟩, ⟨not_q0⟩, ⟨not_q⟩]) = NOCYCLE
                ⊣ prove_goal(⟨not_p0⟩, [⟨not_p⟩, ⟨not_q0⟩, ⟨not_q⟩], {})
                  ⊣ prove_goals([⟨not_q⟩], [⟨not_p0⟩, ⟨not_p⟩, ⟨not_q0⟩, ⟨not_q⟩], {})
                    ⊣ prove_cycle(⟨not_q⟩, [⟨not_p0⟩, ⟨not_p⟩, ⟨not_q0⟩, ⟨not_q⟩])
```

⊣ NegCycle = False
⊣ apply_positive_cycle_rule(⟨*not_q*⟩) = True
⊣ return True
⊣ return (True, {})
⊣ return (True, {(⟨*not_p*$_0$⟩,True)})
⊣ prove_goals([], [⟨*not_p*⟩, ⟨*not_q*$_0$⟩, ⟨*not_q*⟩], {(⟨*not_p*$_0$⟩,True)})
⊣ return (True, {(⟨*not_p*$_0$⟩,True)})
⊣ return (True, {(⟨*not_p*⟩,True), (⟨*not_p*$_0$⟩,True)})
⊣ prove_goals([], [⟨*not_q*$_0$⟩, ⟨*not_q*⟩], {(⟨*not_p*⟩,True), (⟨*not_p*$_0$⟩,True)})
⊣ return (True, {(⟨*not_p*⟩,True), (⟨*not_p*$_0$⟩,True)})
⊣ return (True, {(⟨*not_p*⟩,True), (⟨*not_p*$_0$⟩,True)})
⊣ return (True, {(⟨*not_q*$_0$⟩,True), (⟨*not_p*⟩,True), (⟨*not_p*$_0$⟩,True)})
⊣ prove_goals([], [⟨*not_q*⟩], {(⟨*not_q*$_0$⟩,True), (⟨*not_p*⟩,True),
(⟨*not_p*$_0$⟩,True)})
⊣ return (True, {(⟨*not_q*$_0$⟩,True), (⟨*not_p*⟩,True), (⟨*not_p*$_0$⟩,True)})
⊣ return (True, {(⟨*not_q*$_0$⟩,True), (⟨*not_p*⟩,True), (⟨*not_p*$_0$⟩,True)})
⊣ return (True, {(⟨*not_q*⟩,True), (⟨*not_q*$_0$⟩,True), (⟨*not_p*⟩,True),
(⟨*not_p*$_0$⟩,True)})
⊣ prove_goals([*r*, ⟨*chk*⟩], [], {(⟨*not_q*⟩,True), (⟨*not_q*$_0$⟩,True), (⟨*not_p*⟩,True),
(⟨*not_p*$_0$⟩,True)})
⊣ prove_cycle(*r*, []) = NOCYCLE
⊣ prove_goal(*r*, [], {(⟨*not_q*⟩,True), (⟨*not_q*$_0$⟩,True), (⟨*not_p*⟩,True),
(⟨*not_p*$_0$⟩,True)})
⊣ prove_goals([*r*], [*r*], {(⟨*not_q*⟩,True), (⟨*not_q*$_0$⟩,True), (⟨*not_p*⟩,True),
(⟨*not_p*$_0$⟩,True)})
⊣ prove_cycle(*r*, [*r*])
⊣ NegCycle = False
⊣ apply_positive_cycle_rule(*r*) = True
⊣ return True
⊣ return (True, {(⟨*not_q*⟩,True), (⟨*not_q*$_0$⟩,True), (⟨*not_p*⟩,True),
(⟨*not_p*$_0$⟩,True)})
⊣ return (True, {(*r*,True), (⟨*not_q*⟩,True), (⟨*not_q*$_0$⟩,True), (⟨*not_p*⟩,True),
(⟨*not_p*$_0$⟩,True)})
⊣ prove_goals([⟨*chk*⟩], [], {(*r*,True), (⟨*not_q*⟩,True), (⟨*not_q*$_0$⟩,True),
(⟨*not_p*⟩,True), (⟨*not_p*$_0$⟩,True)})
⊣ prove_cycle(⟨*chk*⟩, []) = NOCYCLE
⊣ prove_goal(⟨*chk*⟩, [], {(*r*,True), (⟨*not_q*⟩,True),
(⟨*not_q*$_0$⟩,True), (⟨*not_p*⟩,True), (⟨*not_p*$_0$⟩,True)})
⊣ prove_goals([], [⟨*chk*⟩], {(*r*,True), (⟨*not_q*⟩,True), (⟨*not_q*$_0$⟩,True),
(⟨*not_p*⟩,True), (⟨*not_p*$_0$⟩,True)})
⊣ return (True, {(*r*,True), (⟨*not_q*⟩,True), (⟨*not_q*$_0$⟩,True),
(⟨*not_p*⟩,True), (⟨*not_p*$_0$⟩,True)})
⊣ return (True, {(⟨*chk*⟩,True), (*r*,True), (⟨*not_q*⟩,True), (⟨*not_q*$_0$⟩,True),
(⟨*not_p*⟩,True), (⟨*not_p*$_0$⟩,True)})
⊣ prove_goals([], [], {(⟨*chk*⟩,True), (*r*,True), (⟨*not_q*⟩,True),
(⟨*not_q*$_0$⟩,True), (⟨*not_p*⟩,True), (⟨*not_p*$_0$⟩,True)})

```
        ⊣ return (True, {(⟨chk⟩,True), (r,True), (⟨not_q⟩,True), (⟨not_q0⟩,True),
                                                    (⟨not_p⟩,True), (⟨not_p0⟩,True)})
      ⊣ return (True, {(⟨chk⟩,True), (r,True), (⟨not_q⟩,True), (⟨not_q0⟩,True),
                                                    (⟨not_p⟩,True), (⟨not_p0⟩,True)})
    ⊣ return (True, {(⟨chk⟩,True), (r,True), (⟨not_q⟩,True), (⟨not_q0⟩,True),
                                                    (⟨not_p⟩,True), (⟨not_p0⟩,True)})
  ⊣ return (True, {(⟨chk⟩,True), (r,True), (⟨not_q⟩,True), (⟨not_q0⟩,True),
                                                    (⟨not_p⟩,True), (⟨not_p0⟩,True)})
⊣ SUCCESS with partial model {(⟨chk⟩,True), (r,True), (⟨not_q⟩,True), (⟨not_q0⟩,True),
                                                    (⟨not_p⟩,True), (⟨not_p0⟩,True)}
```

Now, consider program 1. We will show executions for well founded and stable model semantics. Notice the symmetry with how positive cycles are handled.

Example B.2. Program 1

```
%Origial Rules
p :- s.
p :- not q.
q :- not p.
r :- p.
%Dual rules
⟨not_p0⟩ :- not s.
⟨not_p1⟩ :- q.
not p :- ⟨not_p0⟩,⟨not_p1⟩.

⟨not_q0⟩ :- p.
not q :- ⟨not_q0⟩.

⟨not_r0⟩ :- not p.
not r :- ⟨not_r0⟩.

⟨chk⟩.
```

Well-Founded:

```
⊣ query([p])
  ⊣ prove_goals([p], [], {})
    ⊣ prove_cycle(p, []) = NOCYCLE
    ⊣ prove_goal(p, [], {})
      ⊣ prove_goals([s], [p], {})
        ⊣ prove_cycle(s, [p]) = NOCYCLE
        ⊣ prove_goal(s, [p], {})
          ⊣ return (False, {})
        ⊣ return (False, {})
      ⊣ prove_goals([⟨not_q⟩], [p], {})
        ⊣ prove_cycle(⟨not_q⟩, [p]) = NOCYCLE
        ⊣ prove_goal(⟨not_q⟩, [p], {})
          ⊣ prove_goals([⟨not_q0⟩], [⟨not_q⟩, p], {})
```

```
          ⊣ prove_cycle(⟨not_q0⟩, [⟨not_q⟩, p]) = NOCYCLE
          ⊣ prove_goal(⟨not_q0⟩, [⟨not_q⟩, p], {})
            ⊣ prove_goals([p], [⟨not_q0⟩, ⟨not_q⟩, p], {})
              ⊣ prove_cycle(p, [⟨not_q0⟩, ⟨not_q⟩, p])
                ⊣ NegCycle = True
                ⊣ apply_even_cycle_rule(p) = ⊥
                ⊣ return ⊥
              ⊣ return (⊥, {})
            ⊣ return (⊥, {(⟨not_q0⟩,⊥)})
          ⊣ prove_goals([], [⟨not_q⟩, p], {(⟨not_q0⟩,⊥)})
            ⊣ return (True, {(⟨not_q0⟩,⊥)})
          ⊣ return (⊥, {(⟨not_q0⟩,⊥)})
        ⊣ return (⊥, {(⟨not_q⟩,⊥), (⟨not_q0⟩,⊥)})
      ⊣ prove_goals([], [p], {(⟨not_q⟩,⊥), (⟨not_q0⟩,⊥)})
        ⊣ return (True, {(⟨not_q⟩,⊥), (⟨not_q0⟩,⊥)})
      ⊣ return (⊥, {(⟨not_q⟩,⊥), (⟨not_q0⟩,⊥)})
    ⊣ return (⊥, {(p,⊥), (⟨not_q⟩,⊥), (⟨not_q0⟩,⊥)})
  ⊣ prove_goals([], [], {(p,⊥), (⟨not_q⟩,⊥), (⟨not_q0⟩,⊥)})
    ⊣ return (True, {(p,⊥), (⟨not_q⟩,⊥), (⟨not_q0⟩,⊥)})
  ⊣ return (⊥, {(p,⊥), (⟨not_q⟩,⊥), (⟨not_q0⟩,⊥)})
⊣ SUCCESS with partial model {(p,⊥), (⟨not_q⟩,⊥), (⟨not_q0⟩,⊥)}
```

Stable Models:

```
⊣ query([p])
  ⊣ prove_goals([p, ⟨chk⟩], [], {})
    ⊣ prove_cycle(p, []) = NOCYCLE
    ⊣ prove_goal(p, [], {})
      ⊣ prove_goals([s], [p], {})
        ⊣ prove_cycle(s, [p]) = NOCYCLE
        ⊣ prove_goal(s, [p], {})
          ⊣ return (False, {})
        ⊣ return (False, {})
      ⊣ prove_goals([⟨not_q⟩], [p], {})
        ⊣ prove_cycle(⟨not_q⟩, [p]) = NOCYCLE
        ⊣ prove_goal(⟨not_q⟩, [p], {})
          ⊣ prove_goals([⟨not_q0⟩], [⟨not_q⟩, p], {})
            ⊣ prove_cycle(⟨not_q0⟩, [⟨not_q⟩, p]) = NOCYCLE
            ⊣ prove_goal(⟨not_q0⟩, [⟨not_q⟩, p], {})
              ⊣ prove_goals([p], [⟨not_q0⟩, ⟨not_q⟩, p], {})
                ⊣ prove_cycle(p, [⟨not_q0⟩, ⟨not_q⟩, p])
                  ⊣ NegCycle = True
                  ⊣ apply_even_cycle_rule(p) = True
                  ⊣ return True
                ⊣ return (True, {})
              ⊣ return (True, {(⟨not_q0⟩,True)})
            ⊣ prove_goals([], [⟨not_q⟩, p], {(⟨not_q0⟩,True)})
              ⊣ return (True, {(⟨not_q0⟩,True)})
            ⊣ return (True, {(⟨not_q0⟩,True)})
```

```
p :- not p.
q.
```

Program 4. Odd Cycle Example

```
        ⊣ return (True, {(⟨not_q⟩,True), (⟨not_q0⟩,True)})
      ⊣ prove_goals([], [p], {(⟨not_q⟩,True), (⟨not_q0⟩,True)})
        ⊣ return (True, {(⟨not_q⟩,True), (⟨not_q0⟩,True)})
      ⊣ return (True, {(⟨not_q⟩,True), (⟨not_q0⟩,True)})
    ⊣ return (True, {(p,True), (⟨not_q⟩,True), (⟨not_q0⟩,True)})
  ⊣ prove_goals([⟨chk⟩], [], {(p,True), (⟨not_q⟩,True), (⟨not_q0⟩,True)})
    ⊣ prove_cycle(⟨chk⟩, []) = NOCYCLE
    ⊣ prove_goal(⟨chk⟩, [], {(p,True), (⟨not_q⟩,True), (⟨not_q0⟩,True)})
      ⊣ prove_goals([], [⟨chk⟩], {(p,True), (⟨not_q⟩,True), (⟨not_q0⟩,True)})
        ⊣ return (True, {(p,True), (⟨not_q⟩,True), (⟨not_q0⟩,True)})
      ⊣ return (True, {(⟨chk⟩,True), (p,True), (⟨not_q⟩,True), (⟨not_q0⟩,True)})
    ⊣ prove_goals([], [], {(⟨chk⟩,True), (p,True), (⟨not_q⟩,True),
                                                    (⟨not_q0⟩,True)})
      ⊣ return (True, {(⟨chk⟩,True), (p,True), (⟨not_q⟩,True), (⟨not_q0⟩,True)})
    ⊣ return (True, {(⟨chk⟩,True), (p,True), (⟨not_q⟩,True), (⟨not_q0⟩,True)})
  ⊣ return (True, {(⟨chk⟩,True), (p,True), (⟨not_q⟩,True), (⟨not_q0⟩,True)})
⊣ SUCCESS with partial model {(⟨chk⟩,True), (p,True), (⟨not_q⟩,True), (⟨not_q0⟩,True)}
```

Finally we consider programs with odd cycles. For this example we will make use of well founded and stable model semantics.

Example B.3. Program 4

```
p :- not p.
q.
⟨not_p0⟩ :- p.
not p :- ⟨not_p0⟩.
⟨chk_p0⟩ :- p.
⟨chk⟩ :- ⟨chk_p0⟩.
```

Well-Founded:

```
⊣ query([q])
  ⊣ prove_goals([q], [], {})
    ⊣ prove_cycle(q, []) = NOCYCLE
    ⊣ prove_goal(q, [], {})
      ⊣ prove_goals([], [q], {})
        ⊣ return (True, {})
      ⊣ return (True, {(q,True)})
    ⊣ prove_goals([], [], {(q,True)})
      ⊣ return (True, {(q,True)})
    ⊣ return (True, {(q,True)})
  ⊣ SUCCESS with partial model {(q,True)}
```

Stable Models:

```
⊣ query([q])
```

```
⊣ prove_goals([q, ⟨chk⟩], [], {})
  ⊣ prove_cycle(q, []) = NOCYCLE
  ⊣ prove_goal(q, [], {})
    ⊣ prove_goals([], [q], {})
      ⊣ return (True, {})
    ⊣ return (True, {(q,True)})
  ⊣ prove_goals([⟨chk⟩], [], {(q,True)})
    ⊣ prove_cycle(⟨chk⟩, []) = NOCYCLE
    ⊣ prove_goal(⟨chk⟩, [], {(q,True)})
      ⊣ prove_goals([⟨chk_p0⟩], [⟨chk⟩], {(q,True)})
        ⊣ prove_cycle(⟨chk_p0⟩, [⟨chk⟩]) = NOCYCLE
        ⊣ prove_goal(⟨chk_p0⟩, [⟨chk⟩], {(q,True)})
          ⊣ prove_goals([p], [⟨chk_p0⟩, ⟨chk⟩], {(q,True)})
            ⊣ prove_cycle(p, [⟨chk_p0⟩, ⟨chk⟩]) = NOCYCLE
            ⊣ prove_goal(p, [⟨chk_p0⟩, ⟨chk⟩], {(q,True)})
              ⊣ prove_goals([⟨not_p⟩], [p, ⟨chk_p0⟩, ⟨chk⟩], {(q,True)})
                ⊣ prove_cycle(⟨not_p⟩, [p, ⟨chk_p0⟩, ⟨chk⟩]) = NOCYCLE
                ⊣ prove_goal(⟨not_p⟩, [p, ⟨chk_p0⟩, ⟨chk⟩], {(q,True)})
                  ⊣ prove_goals([⟨not_p0⟩], [⟨not_p⟩, p, ⟨chk_p0⟩, ⟨chk⟩], {(q,True)})
                    ⊣ prove_cycle(⟨not_p0⟩, [⟨not_p⟩, p, ⟨chk_p0⟩, ⟨chk⟩]) = NOCYCLE
                    ⊣ prove_goal(⟨not_p0⟩, [⟨not_p⟩, p, ⟨chk_p0⟩, ⟨chk⟩], {(q,True)})
                      ⊣ prove_goals([p], [⟨not_p0⟩, ⟨not_p⟩, p, ⟨chk_p0⟩, ⟨chk⟩],
                                                                          {(q,True)})
                        ⊣ prove_cycle(p, [⟨not_p0⟩, ⟨not_p⟩, p, ⟨chk_p0⟩, ⟨chk⟩])
                          ⊣ apply_odd_cycle_rule(p) = False
                          ⊣ return False
                        ⊣ return (False, {(q,True)})
                      ⊣ return (False, {(q,True)})
                    ⊣ return (False, {})
                  ⊣ return (False, {})
                ⊣ return (False, {})
              ⊣ return (False, {})
            ⊣ return (False, {})
          ⊣ return (False, {})
        ⊣ return (False, {})
      ⊣ return (False, {})
    ⊣ return (False, {})
  ⊣ return (False, {})
⊣ FAIL
```