NAF-BASED LOGIC SEMANTICS:

PROOF-THEORETIC GENERALIZATION AND NON-GROUND EXTENSION

by

Elmer E. Salazar

APPROVED BY SUPERVISORY COMMITTEE:

Gopal Gupta, Chair

Vibhav Gogate

Sriraam Natarajan

Shiyi Wei

NAF-BASED LOGIC SEMANTICS:
PROOF-THEORETIC GENERALIZATION AND NON-GROUND EXTENSION

by

ELMER E. SALAZAR, BS, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

August 2019

ACKNOWLEDGMENTS

I would like to thank Dr. Gopal Gupta for supervising my research, Dr. Kyle Marple for his work on the s(ASP) system and Howard Blair for his help and input, particularly on the subject of completeness. In addition, I would like to thank Sarat Varanasi, Kinjal Basu, Farhad Shakerin, Zhuo Chen, Serdar Erbatur, and Joaquin Arias.

June 2019

NAF-BASED LOGIC SEMANTICS:

PROOF-THEORETIC GENERALIZATION AND NON-GROUND EXTENSION

Elmer E. Salazar, PhD
The University of Texas at Dallas, 2019

Supervising Professor: Gopal Gupta, Chair

There are several semantics based for negation as failure in logic programming. 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. Two major, but related, topics are covered.

Firstly, while being restricted 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.

Secondly, a Method for computing stable models of normal logic programs, i.e., logic programs extended with negation, in the presence of predicates with arbitrary terms will be presented. Such programs need not have a finite grounding, so traditional methods do not apply. This method relies on the use of a non-Herbrand universe, as well as coinduction,

constructive negation and a number of other novel techniques. Using this method, a normal logic program with predicates can be executed directly under the stable model semantics without requiring it to be grounded either before or during execution and without requiring that its variables range over a finite domain. As a result, the method is quite general and supports the use of terms as arguments, including lists and complex data structures.

TABLE OF CONTENTS

ACKNOWLEDGMENTS iv
ABSTRACT v
LIST OF FIGURES x
LIST OF TABLES xi
CHAPTER 1 INTRODUCTION 1
CHAPTER 2 BACKGROUND 8
2.1 The Language 8
2.2 Negation-as-Failure 10
2.3 The Semantics 13
2.3.1 Fitting's 3-Value 13
2.3.2 Well-Founded 13
2.3.3 Stable Models 14
2.3.4 CoStable Models 15
2.4 SLD Resolution 16
2.5 Coinduction 17
2.6 Coinductive Logic programming 18
2.7 Ground Method 18
CHAPTER 3 GENERALIZATION OF COMPLETION SEMANTICS USING PROOF MODELS 23
3.1 Interpretations 23
3.2 Inductive and Coinductive Proofs for Literals 25
3.3 Finitely Computable Programs 31
3.4 Coinductive Proof Sets 31
3.5 Proof Models 34
3.6 Cycles 37
3.7 Proof Model Form 39
CHAPTER 4 GENERALIZATION OF COMPLETION SEMANTICS USING OPERATOR FIXED-POINT 45
4.1 Handling Cycles 45

4.1.1 Cycle Sets . . . 45
4.1.2 Cycle Resolution Functions . . . 47
4.2 Resolution Form . . . 49
4.2.1 Subsumption . . . 52
4.2.2 Correctness of Resolution Form . . . 55
4.3 Resolution Form of Example Semantics . . . 63
4.3.1 Functions to Handle Cycles . . . 63
4.3.2 Proofs of Resolution Forms . . . 71
CHAPTER 5 GENERALIZATION OF COMPLETION SEMANTICS USING GOAL-DIRECTED, TOP-DOWN ALGORITHM . . . 82
5.1 3-value Modified CoSLD Resolution . . . 82
5.2 Restrictions . . . 82
5.3 Preprocessing . . . 83
5.3.1 Internal IDs . . . 83
5.3.2 Dual Rule Generation . . . 84
5.3.3 Consistency Check . . . 84
5.4 The Rules . . . 85
5.5 The Algorithm . . . 86
CHAPTER 6 THE S(ASP) ALGORITHM . . . 99
6.1 Fundamentals . . . 99
6.1.1 Extended Stable Models and the s(ASP) Universe . . . 99
6.1.2 Operational State, Rules, and Transformations . . . 103
6.1.3 Constructive Unification and Disunification . . . 108
6.1.4 Runtime Restrictions . . . 117
6.2 Advanced Mechanisms . . . 118
6.2.1 Modified CoSLD Resolution . . . 118
6.2.2 Even Loops . . . 124
6.2.3 Consistency Check . . . 126
6.3 The Complete Method . . . 128

6.4 Understanding the Partial Model . . . 133
CHAPTER 7 PROOFS AND APPLICATIONS FOR S(ASP) . . . 135
7.1 Completeness . . . 135
7.2 Proof of Soundness . . . 136
7.2.1 Review of Splitting Theorem . . . 138
7.2.2 Stripping Unneeded Rules and Body Literals . . . 139
7.2.3 Forall . . . 142
7.2.4 Constructive Coinductive Failure . . . 143
7.2.5 Main Proof . . . 145
7.3 Implementation and Examples . . . 148
7.3.1 Example: N Queens with Lists . . . 148
7.3.2 Example: Hamiltonian Cycle Detection . . . 152
7.4 Applications . . . 153
CHAPTER 8 CONCLUSION . . . 156
8.1 Related Work . . . 156
8.2 Further Work . . . 157
8.3 Conclusion . . . 159
REFERENCES . . . 161
BIOGRAPHICAL SKETCH . . . 166
CURRICULUM VITAE

LIST OF FIGURES

1.1 Commonalities Among Semantics. 3
2.1 Even Cycle. 12
3.1 Positive Cycle. 24
4.1 False Cycle. 51
6.1 Abstracted s(ASP) Meta-interpreter. 132
6.1 Abstracted s(ASP) Meta-interpreter Continued. 133
7.1 N Queens Program with Lists. 149
7.2 A program for Hamiltonian cycle detection with a simple graph included. 151

LIST OF TABLES

5.1 Cycle Resolution Rules for Notable Semantics. 86

CHAPTER 1

INTRODUCTION

Considerable amount of research has been done on adding negation to logic programming over the last 40 years (Apt and Bol, 1994; Minker, 1988). One such method is negation-as-failure (NAF). Negation as failure uses a closed world assumption. Anything that cannot be proven is assumed false. While a horne logic program has an unique minimal model, logic programs using NAF may have multiple models. Different semantics can be used depending on which of these models are desirable. These semantics include Fitting's semantics (Fitting and Ben-Jacob, 1988), the well-founded semantics of Ross, Gelder and Schlipf (Van Gelder et al., 1991), the stable model semantics (Gelfond and Lifschitz, 1988), and many others (Ullman, 1994). Dix (Dix, 1995a,b) has done a systematic study of these semantics, proposing a number of properties that can be used to characterize a semantics.

Chapters 3, 4, and 5 show that various NAF semantics can be elegantly characterized via a combination of induction and coinduction. Induction captures well-founded computations while coinduction captures cyclical, consistent computations. The various semantics use 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 or undefined), 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 can be designed for predicate answer set programming as presented in Chapter 6.

The chapters give the declarative and operational semantics for various semantics of propositional 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.

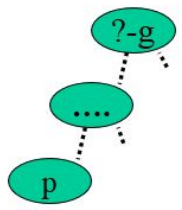
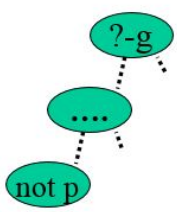
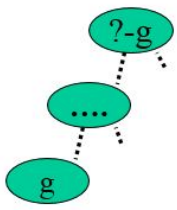
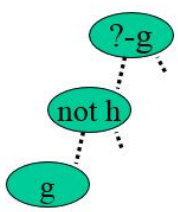
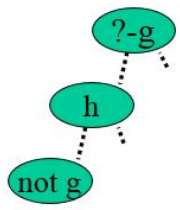
The intuition for this work, loosely speaking, is the following. During execution of a query (a conjunction of goals, each being either a proposition or it's negation) with respect to a logic program, the execution can be well-founded or it can contain cycles that can keep unfolding forever. If the execution is well-founded then all the goals will get resolved during a successful top-down execution of the query consisting of the goal `g`, with the final goal in the final resolvent matching a fact. 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 (Apt and Bol, 1994). 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 (Van Gelder et al., 1991), stable model semantics (Gelfond and Lifschitz, 1988), Fitting's 3-valued semantics (Fitting and Ben-Jacob, 1988), co-stable model semantics (Gupta et al., 2012), 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: $\perp$ (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 $\perp$, 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 $\perp$ (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 1.1.

				
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 $\perp$ Assign True	Many possibilities Assign $\perp$ Assign True Assign False	Assign $\perp$ *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 $\perp$: Fitting True: coSM	$\perp$: Fitting, WFS True: SM, coSM False: SM, coSM	$\perp$: Fitting, WFS SM & coSM will falsify g & h

WFS = Well Founded Semantics; SM = Stable Model Semantics

Figure 1.1: Commonalities Among Semantics.

Chapters 6 and 7 are concerned with stable model semantics, and the direct goal-directed execution of the predicates. Of the various semantics which have been developed, the stable model semantics is widely regarded as the most expressive (Baral, 2003).

However, the current computation methods for stable model semantics and the answer set programming paradigm inspired by them, are computable only for programs which are finitely groundable. Current methods compute the stable models of a program by first *grounding* the program. That is, each program variable is instantiated with each of the values from its respective domain to derive ground clauses, making an instance of the rule (clause) for each combination of variable groundings. The stable models are then computed using the grounded program. In most modern implementations, the ground program is suitably transformed and fed to an ASP solver, which is based on SAT solvers. The models produced by the solver will be the stable models of the original program (Lin and Zhao, 2004; Gebser et al., 2007).

There are several problems with the grounding-based approach. The most significant being that only certain classes of programs are guaranteed to have a finite grounding. A logic program with even a single unary term will have an infinite number of groundings due to the fact that, given a single unary function symbol f and a single constant value `a`, the domain over which variables can range is infinite consisting of {`a, f(a), f(f(a)), f(f(f(a))), ...`}. This results in an infinite number of grounded rules. For such a program to have a finite grounding, each variable must be restricted to a finite domain. However, even with finite domains, the grounding of a program may be exponentially larger than the original.

Secondly, SAT-based and similar approaches compute the complete model of the grounded program. In reality, when solving practical problems, we are often only interested in part of the model. There are times when multiple solutions to a problem can exists in a single model, or only part of the knowledge base represented by the program is needed to answer some query. In the first case, it is often impossible to distinguish which which part of a solution belongs to what solution. To complicate this further it is possible that a part is part of multiple solutions. An example of this would be the Tower of Hanoi problem. Since there

are multiple solution with a minimal number of moves, an asp program (using the traditional approach) needs several lines of code that are simply for restricting what solutions are accepted. The second case can be seen by considering a program that encodes the meanings of words. Most words have multiple meanings, which could be quite different. For instance, is the word tree referring to a plant or a diagram? If the program must reason about the situation to determine which meaning should be used, then all knowledge of both definitions must be in the program, but only one set of knowledge needs to be part of the solution.

Finally, adding negation can lead to (parts of) the program becoming inconsistent. When this occurs, bottom-up methods that work with grounded programs will declare the whole program to be inconsistent (i.e. , no model exists). In practice, it may be desirable to compute answers as long as these answers do not involve the inconsistent part of the program (Marple and Gupta, 2014).

The problems described above can be resolved (or limited in the case of restricting the classes of programs that can be solved) by designing goal-directed or query-driven execution methods for computing stable models. Such a method does not use a SAT solver, but rather, given a query, computes a partial stable model containing the query, if one exists. The computation is done in a manner very similar to SLD resolution (Lloyd, 1987) for logic programs (Marple et al., 2012a; Marple and Gupta, 2014), and an execution method that works with grounded programs has been previously presented (Marple et al., 2012a; Marple and Gupta, 2014). Chapters 6 and 7 build on this previous work to remove the need for grounding, developing a query-driven method that can apply the stable model semantics to a normal logic program containing arbitrary terms as well as negation. This is accomplished without grounding the program, either before or during execution(accept in the case of binding a ground term). It should be noted that, until recently (Marple et al., 2012a), such query-driven procedures were considered impossible to develop even for propositional logic programs (Baral and Gelfond, 1994).

The key insight is to use a non-Herbrand universe (in fact, an infinite *superset* of the Herbrand universe) which allows us to guarantee properties required for the correctness of our method while still obtaining useful results. Additionally, coinductive logic programming is used to establish the consistency of mutually dependent (co-)recursive calls and *dual rules* both simplify the handling of negation and provide constructive negation (Gupta et al., 2007).

The only restrictions the method places upon programs are that operands of arithmetic operations must be ground, two *negatively constrained variables* (discussed in Section 6.1.1) cannot be *disunified* with each other, and unbound recursion does not happen. Of these, the last restriction can be removed via tabling (Swift and Warren, 2012). We prove the soundness of our method for all program runs that satisfy the restrictions and a prototype implementation is available (Marple, 2015). For convenience, we will refer to our method by the name given to its prototype implementation (developed by Kyle Marple): **s(ASP)**.

It should be noted that top-down, goal-directed implementations which support predicates have been designed for the well-founded semantics by extending Prolog systems, for example with *tabling* (Chen et al., 1995). However, the well-founded semantics can be too weak for many applications, as it declares the truth value of many interesting atoms to be unknown. The stable model semantics is more expressive, but to date, there has been no satisfactory solution to the problem of computing stable models of arbitrary predicate logic programs. Those solutions that have been proposed either greatly restrict the types of programs that can be handled or ground the program incrementally during execution (Dal Palù et al., 2009; Dao-Tran et al., 2012; Lefvre and Nicolas, 2009a,b). Thus, our research makes important contributions:

- It presents a top-down, query-driven method that can execute normal logic programs with arbitrary predicates, thus solving a problem that was hitherto considered unsolvable.

- Our method can be thought of as providing an operational semantics to normal logic programs with predicates (or, Prolog with negation) under the stable model semantics. This can be combined with other advanced features of logic programming such as constraints (Jaffar and Lassez, 1987) to develop extremely powerful applications in an elegant manner, such as automated planning under real-time constraints (Bansal et al., 2010).

- The stable model semantics and answer set programming have been shown to support powerful reasoning techniques such as default reasoning, counterfactual reasoning, abductive reasoning, etc. These reasoning capabilities now become available within Prolog.

Chapter 2 presents definitions, explanations, and descriptions that are necessary for understanding the rest of the work. As stated above, Chapters 3, 4, and 5 discuss the generalization of NAF-based semantics, and Chapters 6 and 7 covers the s(ASP) algorithm. Finally, Chapter 8 will provide closing remarks including discussions on future and related work.

CHAPTER 2

BACKGROUND

2.1 The Language

Definition 1. *A* term *is either*

- *a variable,*
- *a constant, or*
- *a compound term of the form* $s(X_1, X_2, \cdots, X_n)$ *where* $n > 0$*, and for all* $1 \leq i \leq n$ X_i *is a term. The functor for the term is* s *and the arity is* n*.*

A structure *is a term that is either a compound term or a constant (which has itself as the functor and arity zero).*

A predicate represents some relation or truth. Like a structure it has a functor an arity. So, references to a predicate can be represented by a structure. Such a representation is called an *instance* of the predicate. Unlike Terms used to represent data, an instance of a predicate can be negated. For convenience, a couple terms will be defined to represent these two things.

Definition 2. *Let* T *be an instance of some predicate. The* T *is called an* atom*, and* ***not*** T *is called its negation. Let* L *be either an atom or its negation.* L *is called a* literal*.*

Definition 3. *A* program *is a set of* rules R *of the following form:*

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

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

In addition, for convenience we define the following functions:

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

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

Definition 4. *A* propositional program *is a program for with every structure in every rule has arity zero.*

Chapters 3, 4, 5 will only be dealing with propositional programs. In fact the semantics considered in this work are defined on propositional programs. A program can be converted to a propositional program by grounding.

Theorem 1. *A program, constraint to some universe, can be rewritten as an equivalent propositional program.*

Proof. Let P be a program, and D be some universe. For all rules $r \in P$, let V_r be the set of variables that appear in r, and A_r be the set of all functions that map all the variables in V_r to some member of D. Given some rule r and $\alpha \in A_r$, A *ground* rule, $\texttt{ground}(r,\alpha)$, can be constructed by replacing each variable V in r with $\alpha(V)$. Let $P' = \{r' \mid r \in P, \alpha \in A_r, \texttt{ground}(r, \alpha)\}$, and S be the set of all structures in P'. Let C be a set of constants such that there is a one-to-one mapping from S to C. Construct program P'' by replacing each structure in P' with its corresponding constant in C.

□

Just as a program be grounded to produce a propositional program, other forms of shortcuts can be defined. Some such syntactic sugar that can be found in galliwasp and s(ASP) are as follows:

The Headless Rule. Let P be some program. Let r be a rule in P of the form "$:-B_1, B_2, \cdots, B_n$." where $n > 0$ and for all $1 \leq i \leq n$, B_i is a literal. r should be rewritten as "p:$-B_1, B_2, \cdots, B_n$,**not** p.", where p is a constant that is not in P.

Classical Negation. An atom beginning with "-" is considered a classical negation, and cannot be true at the same time as the atom gotten by removing the "-". In stable models, for some atom −p, the additional rule :− p,−p should be added.

Abducibles. In stable models for an abducible atom, a even cycle is generated. So if p is abducible, then "p:− **not** n_p. n_p:− **not** p." should be added to the program.

2.2 Negation-as-Failure

It is possible for a program to have many models. Model-theoretic semantics such as the ones used in this document determine which models of a program are wanted. The are also called preferred models. The semantics considered here will be those 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 5. *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 $\mathtt{p} \in props(P)$, let B be a disjunction of conjunctions such that each conjunction in B is the body of some rule in P with $\mathtt{p}$ as the head, and B contains all*

such conjunctions. Then, $\mathtt{p} \iff B$ *is the* completion rule *for* $\mathtt{p}$*. The* completion *of* P *is the set of all such completion rules.*

In addition, we will assume $\bot \iff \bot$ *is true.*

Definition 6. *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 a new symbol `not p` and rules for `not p` so that `not p` is true if and only `p` cannot proven can be added. The resulting program is called the extended program.

Definition 7. *For some program* P*, the* extended program*,* $ext(P)$*, is defined by extending* P *as follows:*

For each proposition $\mathtt{p} \in props(P)$*:*

- *If* $\mathtt{p}$ *is not the head of any rule in* P*, then add a fact for* `not p`*.*
- *If there is a fact for* $\mathtt{p}$ *in* P*, then ignore* $\mathtt{p}$*.*
- *Otherwise, take the body of the Clark's Completion rule for* $\mathtt{p}$*, 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 the program in Figure 2.1. 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. The representation for programs can be extended to account for that.

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

Figure 2.1: Even Cycle.

Definition 8. *Let P be a program. For each rule $r \in ext(P)$ with a negative literal in the head, r is of the form:*

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

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

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

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

As stated earlier, a model-theoretic semantics (semantics for sort) determines which models are preferred. A semantics can be viewed as a function that maps programs to sets of models, and this definition will be used throughout this work.

Definition 9. *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 $\mathcal{S}$.*

2.3 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. Presented below is a brief review of a few semantics, how models are computed for them, and an informal discussion on how the cycles are handled in each case. For this section the traditional definition of an interpretation will be used. That is, Interpretations are sets of propositions with the assumption that any missing propositions are false.

2.3.1 Fitting's 3-Value

Fitting's 3-value semantics (Fitting and Ben-Jacob, 1988) 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 here, but can be found in the original paper (Fitting and Ben-Jacob, 1988).

2.3.2 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 (Van Gelder et al., 1991).

Definition 10. *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 11. $\mathbf{U}_P(I)$*, 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 12. *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.

2.3.3 Stable Models

Stable models uses multiple worlds, rather than assign $\perp$, to stratify the program (Gelfond and Lifschitz, 1988).

Definition 13. *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 14. *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 p is guessed to be 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 p is guessed to be 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.

2.3.4 CoStable Models

CoStable models is a semantics based on stable models presented in the Co-LP 2016 workshop.(Gupta et al., 2012)

Definition 15. *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.*
- *Remove all literals from the body of the rules in the resulting program.*

Definition 16. *For some program P, a set of proposition $I \subseteq props(P)$ 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.

All propositions in an even cycle can be divided into two sets A and B such that p $\in A$ if there exists 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 q $\in A$ that is in the body of the rule part of the even cycle with p as the head. In addition, A and B can be interchanged and the property still holds. By choosing A or B to be a subset of the costable model candidate, two worlds can be created: 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.

2.4 SLD Resolution

SLD resolution is the core of traditional prolog systems. This section will only consider propositional programs, giving a rough overview. More can be learned from the many sources available such as *The art of Prolog* (Sterling and Shapiro, 1994).

SLD resolution is based on resolution theory, and is used to prove some conjunction of atoms, called a query, given a program. traditionally, SLD resolution operates on horne clauses (rules), an this section will assume such. That is there are no negated literals. At the core, the process is simple.

1. Select a atom g from the query, Q.

2. select a rule from the program with a matching head.

3. Replace g in Q with a conjunction of the atoms in the body of the rule.

If the selected rule is a fact, then g will be replaced with nothing. In other words, g is simply removed from Q. If a rule cannot be selected, then go back to the last choice made and make a different one. This process continues until either Q is empty, in which case the query succeeded, or there are no more options to select and the algorithm cannot move forward, in which case the query had failed.

Traditionally, when selecting atoms from the query a atom from the last rule body added to the query is selected. This forms a depth first search through possible proofs of the query.

2.5 Coinduction

The proof-theoretic algorithms presented in this document are based on coinductive logic programming, which is in turn based on coinductuction from category theory (Gupta et al., 2011). Category theory is an abstraction of studies into mathematical abstractions such a groups and rings (Leinster, 2016).

Induction, often seen in computer science in the form of recursive definitions and inductive proofs, allows for proofs of properties or equivalence of algebraic structures. Coinduction allows for proofs over coalagraic structures. Coalgebraic structures cannot be constructed, and can only be deconstructed. One way to view a co algebra is as a black box. It's known what structure it adheres to and information can be extracted based on that structure. As an example, the set of finite lists are algebraic. A finite list can be constructed starting with the empty list. Infinite lists, however are coalgebraic. No infinite list can be constructed from some base case. Instead, it must be known that the list is infinite. It can be deconstructed into a head and a tail, but that tail will still be infinite.

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 (Jacobs and Rutten, 1997).

2.6 Coinductive Logic programming

SLD can be viewed as an inductive proof method based on resolution theory. CoSLD (Simon et al., 2007; Min, 2009) is likewise a coinductive proof method and can be considered a form of circular coinduction (Roşu and Lucanu, 2009). Completion semantics require a combination of induction and coinduction. Kyle Marple presents a modified CoSLD resolution algorithm in order to allow for induction aspects of stable-model semantics (Marple et al., 2012b).

CoSLD resolution modifies SLD resolution by allowing the detection of cycles and reusing previous proofs. Like with SLD resolution, it will be assumed that the rules are horne clauses. When proving a query, both a coinductive hypothesis set (CHS) and a call stack are kept. Whenever a goal is selected from the query (in this case it must be from the last rule body inserted into the query) it is added to the top of the call stack. Whenever a atom is proven (succeeds) it is added to the CHS.

Now, before selecting a rule, the CHS and call stack is checked. If the atom is in the CHS or the callstack, automatically succeed (remove it from the query).

As can be seen, if a cycle exists, the proof cannot be constructed, but it can be deconstructed. So, coinductive logic programming assumes truth. Another way of saying this is that the atom was guessed to be true, and the cycle does not contradict this.

2.7 Ground Method

Now that we have introduced the basics of the stable model semantics, we can discuss the method for goal-directed execution of propositional programs, which can be viewed as a stepping stone to the method for predicate programs. Our method for propositional programs has been proven sound and complete with respect to the Gelfond-Lifschitz method and forms the core of the Galliwasp ASP system (`http://galliwasp.sourceforge.net`) (Marple and Gupta, 2013; Marple, 2014b,a). The two key aspects of the propositional method are its handling of rules containing odd loops over negation and its use of *coinduction*.

Both the propositional and predicate methods categorize rules by examining the call graph and checking the number of negations between any recursive calls.

Definition 17. *A program's **call graph** is a directed, weighted graph with one node for each positive literal in the program. Edges are drawn from rule heads to their goals. While only positive literals are used as nodes, negation is preserved using weighted edges: edges corresponding to a positive literal are given a weight of 0, while edges corresponding to negative literals are given a weight of 1. To keep track of which rules are part of a given cycle, each edge is also paired with an ID indicating which rule produced it.*

First, the call graph is traversed to identify any odd loops over negation.

Definition 18. *An **odd loop over negation (OLON)** occurs when a cycle in the call graph contains an odd number of negations.*

Each rule in the program is then classified using the following definitions:

Definition 19. *An **OLON rule** is a rule which can be called as part of an OLON.*

Definition 20. ***Ordinary rules** have at least one path in the call graph which will not result in an odd loop over negation.*

Note that rules with an empty head are always treated as OLON rules. Additionally, a rule can be both an OLON rule and an ordinary rule via different paths in the call graph.

OLON rules are important to the stable model semantics because they have the ability to place global constraints on a program. These constraints must be satisfied by any stable model, even if the OLON is never reached during execution. Consider the following two forms of OLON rules:

```
p :- B, not p.
:- B.
```

where `B` is some conjunction of goals. For the first rule, any stable model must satisfy one of two cases: (i) `p` is added to the model by another rule in the program, or (ii) at least one goal in `B` must fail. That is, the rule imposes the global constraint `p` $\vee$ `not B`. For headless rules (a shorthand for the second form), the second case must always hold, imposing the global constraint `not B`.

Programs are executed using a modified form of coinduction extended with negation. First, a query is extended to enforce the constraints imposed by any OLON rules in the program. Then, this query is executed using a modified form of coinductive SLD resolution (Gupta et al., 2007).

Definition 21. *Under* ***SLD resolution*** *(Lloyd, 1987), query is executed by calling each goal in turn. Calls are added to the call stack and expanded by selecting clauses whose head unifies with the call and recursively calling the goals in the body. A call succeeds when this expansion becomes empty (the call or its children unify with facts). If no expansion is possible, backtracking occurs: execution is rolled back to the previous expansion operation and the call is expanded using the next matching clause. A call fails when no matching clauses remain. Execution succeeds when every goal of the query has succeeded and fails when both expansion and backtracking are impossible.*

Definition 22. ***Coinductive SLD resolution (co-SLD resolution)*** *expands SLD-resolution by storing each succeeding call in a set called the* ***coinductive hypothesis set (CHS)****. If a call unifies with a call that is already in the CHS, or with an ancestor in the call stack, the call is allowed to coinductively succeed without further expansion (Gupta et al., 2007).*

Under the stable model semantics, the condition for coinductive success via the call stack is modified such that only cycles containing *even loops* may succeed. This modification is necessary because the stable model semantics requires that *positive loops* fail, while traditional coinduction would allow them to succeed.

Definition 23. *An **even loop** occurs when a recursive call is encountered with an even, non-zero number of negations between the call and its ancestor in the call stack.*

Definition 24. *A **positive loop** occurs when a recursive call is encountered with no negations between the call and its ancestor in the call stack.*

This method also add the idea of *coinductive failure*, in which failure and backtracking occur if the negation of a call unifies with a call in the call stack or CHS. This ensures that the CHS remains consistent, as `p` and `not p` can never be present at the same time.

Under this method, the CHS also serves as a *candidate partial model*, or candidate model for simplicity. These are conceptually the same as the candidate stable models, except that the method focuses on finding subsets of stable models rather than complete models (see Definition 26).

Candidate partial models are generated by executing the ordinary rules in a program and then testing to ensure that they satisfy any constraints imposed by OLON rules. This testing is handled by the *non-monotonic reasoning check*.

Definition 25. *The **non-monotonic reasoning check (NMR check)** is a special rule responsible for applying the constraints imposed by OLON rules. A call to the NMR check is automatically appended to each query.*

For each OLON rule in a program, a "sub-check" rule with a unique head is created to apply the corresponding global constraint. The head of each sub-check rule is then added to the body of the NMR check. Sub-check rules are created by adding the negation of the corresponding OLON rule's head to the body (if not already present) and then negating the rule. Each clause is processed independently, so no modification is needed if a goal appears in multiple OLONs or as the head of multiple OLON rules. For instance, the rules

```
p :- B, not p.
```

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

would produce the sub-check rules

```
chk_p1 :- not B.
chk_p1 :- p.
chk_p2 :- q.
chk_p2 :- p.
chk_p3 :- not q.
chk_p3 :- not r.
chk_p3 :- p.
```

As a result, if the NMR check succeeds, the candidate partial model in the CHS must satisfy every OLON rule in the program. Correspondingly, if a program or candidate partial model is inconsistent, the NMR check will trigger failure and backtracking. For example, a program containing the rule

```
:- not c.
```

where `c` does not appear anywhere else in the program will have no stable model. The method enforces this by creating the NMR sub-check

```
chk :- c.
```

Since the program contains no rules for `c`, the check is unsatisfiable and execution will eventually fail.

Upon successful execution of both the query and NMR check, the CHS will be returned as a partial stable model.

Definition 26. *A **partial stable model** is a set of literals which is guaranteed to be a subset of some stable model of the program (Marple et al., 2012a).*

CHAPTER 3

GENERALIZATION OF COMPLETION SEMANTICS USING PROOF MODELS

In the Chapter 2 it was 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. This was shown informally for several semantics. This section we will formally define and prove this claim. Please note that the programs in this chapter (as well as in Chapters 4 and 5 will be assumed to be propositional.

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.

3.1 Interpretations

For the rest of this chapter as well as Chapters 4 and 5, 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 27. *For some program P, a set of literals, $I \subseteq lit(P)$, is called an* interpretation *for P, and for each proposition, p:*

- *if $p \in I$ and* ***not*** *$p \notin I$ then p is true in I.*

- *if* ***not*** $p \in I$ *and* $p \notin I$ *then* p *is false in* I*.*
- *if* p, ***not*** $p \notin I$ *then* p *is unknown (or* $\perp$ *) in* I*.*
- *if* p, ***not*** $p \in I$ *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.

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

Figure 3.1: Positive Cycle.

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

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

Example 1 (Program 3.1)**.** *For the interpretation* $I = \{p, q\}$*,* p *is supported by* I*, but* r *is not.*

Definition 29. *For some program* P*, literal* L *is* supported as unknown by *interpretation* I *if and only if there exists some rule in* $ext(P)$ *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 2 (Program 3.1)**.** *Continuing from the last example with interpretation $I = \{p, q\}$, r is not supported, but it is supported as unknown.*

3.2 Inductive and Coinductive Proofs for Literals

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 30. *Let P be a normal logic program, and L be a literal in $lit(P)$. 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 $ext(P)$, then Π_L contains a single node with label L.*
- *Otherwise, if there exists a rule in $ext(P)$ 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 2. *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 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 31. *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)* $\in \{true, \bot\}$*, 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 32. *A literal L of a program P, is called* coinductive *if and only if neither L nor its negation are inductive.*

Theorem 3. *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 $L \in \text{lit}(P)$ is coinductive then

1. for all rules $r \in \text{ext}(P)$ with $\text{head}(r) = L$ there exists some $L' \in \text{body}(r)$ 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, and 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 33. *Let c be a coinductive proof. Then we can construct a function ϕ_c such that*

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

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

3.3 Finitely Computable Programs

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 34. *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 $L \in lit(P)$ 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.

3.4 Coinductive Proof Sets

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

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

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

- *$root(c) = root(c')$ and $label(c) \neq label(c')$,*
- *$root(c)$ is the negation of $root(c')$ and $label(c) \neq$ **not** $label(c')$, or*
- *$\exists c'' \in support(c)$ 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 37. *Let c_1, c_2 be coinductive proofs. We say c_1 and c_2 are* equivalent *if and only if $root(c_1) = root(c_2)$ and $label(c_1) = label(c_2)$. 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 38. *A coinductive proof set C for some program P is called* complete *if and only if for all propositions $p \in lit(P)$ 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 39. *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 $Sup(C_1)$ covers c, or*
- *C_2 is larger than $C_1 \cup Sup(C_1)$.*

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

Definition 41. *Let C be a complete coinductive proof set. Then, C is* invalid *if and only if there exists $c \in C$ with $label(c) = \bot$ and there exists a coinductive proof c' supported by C such that $label(c') = true$ and either $root(c')$ or **not** $root(c')$ is $root(c)$.*

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.

3.5 Proof Models

Definition 42. *For some program P, a* proof model *of P is a largest coinductive proof set for P that is complete, consistent, and valid.*

Lemma 1. *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:

$\mathbf{h} \in \mathbf{I}$. 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 $\in$ 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 $\notin$ 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 4. *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 1, 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 $L, \textbf{not}\ L \notin M$, 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) = $\bot$. 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 $L', \textbf{not}\ L' \notin M$). 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 $L, \textbf{not}\ L \notin M$ there exists a rule with L as the head and for all L' in the body $L' \in M$ or $L', \textbf{not}\ L' \notin M$(and there is at least one). So, we may construct a coinductive proof with L as the root, $\bot$ 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 $L, \textbf{not}\ L \notin M$. 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 $L, \textbf{not } L \notin M$, 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 43. *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 44. *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 45. *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.*

3.6 Cycles

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 46. *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 $root(c) = L_0$ and $\exists c' \in support(c)$ such that $L_1, L_2, \ldots, L_n, L_0$ is a direct cycle of c'.*

There is an associated value of $label(c)$.

Definition 47. *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*

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

Definition 48. *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 2. *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 Hypothesis. 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 49. *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 $L_i =$ **not** L_j, then the cycle is called an* odd cycle.
- *Otherwise the cycle is an* even cycle

3.7 Proof Model Form

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 50. *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 $\tau : A \to \{true, \bot\}$ such that $\bigwedge_{a \in A} (\tau(a) \in f(a,c,P) \wedge \tau(a)) = label(c)$.*

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 51. *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, $\mathcal{S}(P) = \{M \mid M' \in \mathcal{P}(P), M = model(M)\}$ then we say that $\mathcal{P}$ is the proof model form *of $\mathcal{S}$.*

Theorem 5. *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 4, 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. *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, $\bigwedge_{a \in A} \tau(a) = label(c)$.*

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 2.

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 6. *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 4, 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 $\tau : A \to \{\text{true}, \bot\}$ 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), $\bigwedge_{a \in A} (\tau(a) \in f(a, c, P) \wedge \tau(a)) \neq$ label(c). Since for all a, $\tau(a)$ is the associated value in c, by lemma 3 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. $\square$

CHAPTER 4

GENERALIZATION OF COMPLETION SEMANTICS USING OPERATOR FIXED-POINT

As with theorem 6, 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.

4.1 Handling Cycles

4.1.1 Cycle Sets

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

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 53. *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 Chapter 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 chapter 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 54. *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 $p \in neg(r)$.*

Definition 55. *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 $p \in pos(r)$.*

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 56. *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 57. *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 58. *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.

4.1.2 Cycle Resolution Functions

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

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

Each cycle set paired with a cycle resolution can be associated with a cycle. We simply list the propositions (or their negations) in the head of the rule in the cycle set in order assignment them true or $\bot$, based on the cycle resolution.

Theorem 7. *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 head$(r_0) =$ prop(L_0).
- For $0 < i \leq n$, choose a rule $r_i \in C$ such that head$(r_i) =$ prop(L_i), **not** $L_i \in R$, and if L_{i-1} is negated then **not** $L_i \in$ body(r_{i-1}) otherwise $L_i \in$ 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 60. *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 61. *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 62. *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 functions for convenience. These functions will be used to extract the unresolved cycle sets in a program with respect to some interpretations.

Definition 63. *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.

4.2 Resolution Form

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 64. *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* $\texttt{G}(C) \subseteq C$ *for some* C*. Then, the 4-tuple (*`PC, EC, L, G`*) is called a semantics in* resolution form*.*

Our goal of this chapter 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 65. *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 66. *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. *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(π') $\in \mathbf{T'_p}(I)$.

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 $L \in \mathbf{T'_p}(I)$. □

The $\mathbf{T'_p}$ 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 $\mathbf{T'_p}$ 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 Figure 4.1.

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

Figure 4.1: False Cycle.

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 67. *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 3 (Program 3.1)**.** *Suppose we have an interpretation $\{p, q, r, \textbf{not } p, \textbf{not } q, \textbf{not } r\}$. Then, both $\{\{p, q\}, \{r\}\}$ and $\{\{\textbf{not } p, \textbf{not } q\}, \{r\}\}$ 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. This function will be paramaterized with the functions that determine how cycles are handled, and will compute

a set of models. Internally, there is a function that "steps" through a set of interpretation, resolving some of the literals. The herbrand fixed-point of this function is then the canidate set of models. Unwanted models are then filtered out, resulting the the set of models for the program.

Definition 68. *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 $\exists I'' \in I, \exists A \in \mathtt{PC}(I''), \exists B \in \mathtt{EC}(I'')$, 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 8. *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 $\exists I'' \in I. I' = (\mathbf{T}'_\mathbf{p})^\infty(\mathtt{L}(I''))$ and I' has no unresolved propositions.*

$\mathtt{G}(\mathcal{F}(hfp(\mathbf{T}_\mathbf{P}^\mathcal{S})))$ is the set of all models of P with respect to $\mathcal{S}$.

To prove theorem 8 we need to show three things. We must show that the herbrand fixed-point exists for $\mathbf{T}_\mathbf{P}^\mathcal{S}$, that $\mathtt{G}(\mathcal{F}(\text{hfp}(\mathbf{T}_\mathbf{P}^\mathcal{S})))$ 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 $\mathtt{G}(\mathcal{F}(\text{hfp}(\mathbf{T}_\mathbf{P}^\mathcal{S})))$ as the set models for a program with respect to the semantics.

4.2.1 Subsumption

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 69. *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*
- $\forall L \in I_2, \textbf{\textit{not}}\ L \notin I_2 \Rightarrow L \in I_1$.

Theorem 9. *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 70. *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 5. *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 $\mathbf{T'_p}(I)$. Then, if $\mathbf{T'_p}(I) \sqsupseteq I$ then $\mathbf{T'_p}(\mathbf{T'_p}(I) \setminus R) \sqsupseteq \mathbf{T'_p}(I)$.*

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

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

Case 2: $\exists L \in \mathbf{T'_p}(I)$ such that **not** $L \notin \mathbf{T'_p}(I)$, but $L \notin \mathbf{T'_p}(\mathbf{T'_p}(I) \setminus R)$. Then for all rules in ext(P) with L as the head there exists at least one body literal that is not in $\mathbf{T'_p}(I) \setminus R$. 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 6. *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 5, 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 10. *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 $\text{lit}\,(P)$. Therefore, $\mathbf{T}^{\mathcal{S}}_{\mathbf{P}}(\{\,\text{lit}\,(P)\}) \sqsupseteq \{\,\text{lit}\,(P)\}$ and by lemma 6 each step subsumes the previous. Thus the herbrand fixed-point of $\mathbf{T}^{\mathcal{S}}_{\mathbf{P}}$ exists. □

4.2.2 Correctness of Resolution Form

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

Lemma 1 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 $\mathtt{G}(\mathcal{F}(\text{hfp}(\mathbf{T}_\mathbf{P}^\mathcal{S})))$ to show that they are all models.

Lemma 7. *Let $\mathcal{S} = (\mathtt{PC},\mathtt{EC},\mathtt{L},\mathtt{G})$ be a semantics in resolution form, and P be a program. Then, $\mathtt{G}(\mathcal{F}(\mathit{hfp}(\mathbf{T}_\mathbf{P}^\mathcal{S})))$ is a set of models for P.*

Proof. Since $\mathtt{G}(\mathcal{F}(\text{hfp}(\mathbf{T}_\mathbf{P}^\mathcal{S}))) \subseteq \mathcal{F}(\text{hfp}(\mathbf{T}_\mathbf{P}^\mathcal{S}))$, we only need to prove $\mathcal{F}(\text{hfp}(\mathbf{T}_\mathbf{P}^\mathcal{S}))$ is a set of models. Let $M = \mathcal{F}(\text{hfp}(\mathbf{T}_\mathbf{P}^\mathcal{S}))$. From lemma 1, 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 $p, \mathbf{not}\ p \notin I$, 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 $\mathbf{T}'_\mathbf{p}$ operator. Thus for all $L \in I$ we know $L \in \mathbf{T}'_\mathbf{p}(I)$, and by the definition of $\mathbf{T}'_\mathbf{p}$, there must exist a rule in 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 $\perp$ literals then, property 3 trivially holds. Therefore, assume there is at least one proposition such that it is $\perp$ in I. For all such propositions, p, we know that neither p nor **not** p are inductive. Otherwise by lemma 4 either p or **not** p would be in I. Thus, they must be coinductive. Let $I = \mathbf{T'_P}^{\infty}(\mathtt{L}(I'))$ for some interpretation I'. There are two possible ways p could become $\perp$ 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 $\mathbf{T'_P}^{\infty}(\mathtt{L}(I')$ 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 $\perp$ in I'. Then, there must exist an interpretation I_2 generated while computing the Herbrand fixed-point of $\mathbf{T}_{\mathbf{P}}^{\mathcal{S}}$ 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 8. *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 71. *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, $cycle(C, R, L) = L_1, L_2, \ldots, L_n$ 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 $head(r) = L_i$ and $L_{i+1} \in body(r)$ otherwise there exists a rule $r \in C$ such that $head(r) = prop(L_i)$ and **not** $L_{i+1} \in body(r)$.*

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

With these we can now prove lemma 8.

Proof of lemma 8. 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 $\mathcal{C}^+$ and $\mathcal{C}^-$ respectively. That is,

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

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 $\{\text{true}\ , \bot\ \}$, and for all $x \in X$, some $c \in \mathcal{C}(I)$ 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`.

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

$\mathtt{L}$ 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 $\mathtt{L}$ will be thrown away by the $\mathcal{F}$ function defined in theorem 8. 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 $\mathcal{I}(I) = (\mathbf{T}'_{\mathbf{p}})^{\infty}(I \setminus \{L \mid c \in \mathcal{C}(I), c \text{ is odd}, R \subseteq I, \{L, \mathbf{not}\ L\} \subseteq R, R \text{ resolves } c, F(R, \mathtt{C}_{f_o}) \vee F(R, \mathtt{L}_{f_o})\})$.

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, $\mathtt{L}(I) = \mathcal{I}(I) \cup \{\mathbf{not}\ L \mid L \in \mathcal{I}(I), c \in \mathcal{C}(\text{lit}(P)), \neg \cdot R \subseteq \mathcal{I}(I), \mathbf{not}\ L \in R, R \text{ resolves } c, \neg F'_L(R)\}$.

Finally, we define $\mathtt{G}$.

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

Now we must show that a model is in $\mathtt{G}(\mathcal{F}(\text{hfp}(\mathbf{T}_{\mathbf{P}}^{\mathcal{S}'})))$ if and only if the equivalent proof model is accepted by $\mathcal{S}$. Let $\mathcal{M}_P = \mathtt{G}(\mathcal{F}(\text{hfp}(\mathbf{T}_{\mathbf{P}}^{\mathcal{S}'})))$. 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 50, 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 2.

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 $\text{model}(M) \notin \mathcal{M}_P$. There are four places $\text{model}(M)$ could be eliminated when computing the models.

1. $\text{model}(M)$ could have been computed, but then removed by `G`,

2. $\text{model}(M) \in \text{hfp}(\mathbf{T}_\mathbf{P}^{\mathcal{S}'})$ 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 $R \subseteq \text{model}M$ 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 $\neg \cdot R \subseteq \text{model}M$ 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 2 and 2 we know that the value of p must depend on a direct cycle. This means there is no cycle resolution in $\texttt{PC}(M')$ or $\texttt{EC}(M')$ 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 8. Proof follows from lemma 7 and lemma 8. □

Now we will take a closer look at using resolution form to define a semantics. From the proof for lemma 8, 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 $\perp$.

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

To see how cycle sets and resolutions are computed, consider Figure 3.1. 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.

4.3 Resolution Form of Example Semantics

In this section, we will take a look at the example semantics in Section 2.3. First the behaviour will be broken up into how cycles are handled. For each type of cycle and behavior a function is created to model it. THen these functions are used to parameterize the semantics and proofs are given for the correctness of the parameterization.

4.3.1 Functions to Handle Cycles

We will define some useful functions that can be used to define the semantics presented in our background chapter. 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 $\perp$, 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 $\perp$ 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 73. *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 $\mathcal{C}^+(I)$. Then, let $\mathcal{P}_P^-$ be a positive cycle resolution function such that $\mathcal{P}_P^-(I) = \{R\}$*

Example 4 (Program 3.1)**.** *If $I = \{p, q, r, \textbf{\textit{not}}\ p, \textbf{\textit{not}}\ q, \textbf{\textit{not}}\ r\}$ then $\mathcal{P}_P^-(I) = \{\{\{p, q\}, \{r\}\}\}$.*

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

Theorem 11. *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 7, 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 74. *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 $\mathcal{C}^+(I)$. Then, let $\mathcal{P}_P^+$ be a positive cycle resolution function such that $\mathcal{P}_P^+(I) = \{R\}$*

Example 5 (Program 3.1)**.** *If* $I = \{p,\ q,\ r,\ \textbf{\textit{not}}\ p,\ \textbf{\textit{not}}\ q,\ \textbf{\textit{not}}\ r\}$ *then* $\mathcal{P}_P^+(I) = \{\{\{\textbf{\textit{not}}\ p,\ \textbf{\textit{not}}\ q\},\ \{\textbf{\textit{not}}\ r\}\}\}$.

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

Theorem 12. *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 7, 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 $\perp$ to positive cycles. This would be used, for example, when defining Fitting's 3-value semantics.

Definition 75. *Let* P *be a program and* I *an interpretation of* P*. Let* R *be a set of positive cycle resolutions such that* $\forall r \in R \forall L \in r.\textbf{\textit{not}}\ L \in r$*, and* R *resolves* $\mathcal{C}^+(I)$*. Then, let* $\mathcal{P}_P^=$ *be a positive cycle resolution function such that* $\mathcal{P}_P^=(I) = \{R\}$

Example 6 (Program 3.1)**.** *If* $I = \{p,\ q,\ r,\ \textbf{\textit{not}}\ p,\ \textbf{\textit{not}}\ q,\ \textbf{\textit{not}}\ r\}$ *then* $\mathcal{P}_P^=(I) = \{\{\{p,\ \textbf{\textit{not}}\ p,\ q,\ \textbf{\textit{not}}\ q\},\ \{r,\ \textbf{\textit{not}}\ r\}\}\}$.

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

Theorem 13. *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 7, 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 76. *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 $\forall R' \in R$, $\forall r \in R', L \in r \Rightarrow$ **not** $L \notin r$ and R' resolves $\mathcal{C}^+(I)$. Then, let $\mathcal{P}_P^*$ be a positive cycle resolution function such that $\mathcal{P}_P^*(I) = R$.*

Theorem 14. *For all programs P, proof models M of P, and positive cycles C for M, if a resolution form for a semantics uses $\mathcal{P}_P^*$, $f_p(C, M, P) = \{true\}$.*

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 $\mathcal{C}^+(I)$ 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 77. *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 $\mathcal{C}^+(I)$. Then, let $\mathcal{P}_P$ be a positive cycle resolution function such that $\mathcal{P}_P(I) = R$.*

Theorem 15. *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 $\mathcal{C}^+(I)$ 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 /$\bot$ and false /$\bot$. 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 78. *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 $\mathcal{C}^-(I)$. Then, let $\mathcal{E}_P^{WF}$ be an even cycle resolution function such that $\mathcal{E}_P^{WF}(I) = \{R\}$.*

Example 7 (Program 2.1)**.** *Suppose we have an interpretation* $I = \{p,\ q,\ r,\ s,\ \textbf{not}\ p,\ \textbf{not}\ q,\ \textbf{not}\ r,\ \textbf{not}\ s\}$*. Then,* $\mathcal{E}_P^{WF}(I) = \{\{\{p,\ q,\ \textbf{not}\ p,\ \textbf{not}\ q\}\}\}$*.*

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

Theorem 16. *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 7, 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 79. *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* $\forall R' \in R, \forall r \in R', \forall L \in r.\textbf{not}\ L \notin r$*, and* $\forall R' \in R$*,* R' *resolves* $\mathcal{C}^-(I)$*. Then, let* $\mathcal{E}_P^{SM}$ *be an even cycle resolution function such that* $\mathcal{E}_P^{SM}(I) = R$

Example 8 (Program 2.1)**.** *Suppose we have an interpretation* $I = \{p,\ q,\ r,\ s,\ \textbf{not}\ p,\ \textbf{not}\ q,\ \textbf{not}\ r,\ \textbf{not}\ s\}$*. Then,* $\mathcal{E}_P^{SM}(I) = \{\{\{p, \textbf{not}\ q\}\},\ \{\{q, \textbf{not}\ p\}\}\}$*.*

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

Theorem 17. *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 7, 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 8 does not make this assumption.

Definition 80. *For an interpretation I, $\mathcal{L}^{WF}(I) = \{L : L \in I, \textbf{\textit{not}}\ L \notin I\}$*

$\mathcal{L}^{WF}$ 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 18. *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 $\mathbf{T}_\mathbf{P}^\mathcal{S}$ 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 7, 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 $\mathbf{T}^{\mathcal{S}}_{\mathbf{P}}$ it will be eliminated as a possible model. This is for semantics such a stable models that cannot have odd cycles.

Definition 81. *For an interpretation I, $\mathcal{L}^{SM}(I) = I$.*

Theorem 19. *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 $\mathbf{T}^{\mathcal{S}}_{\mathbf{P}}$ 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 82. *For a set of interpretations I, $\mathcal{G}(I) = I$.*

4.3.2 Proofs of Resolution Forms

Using different combinations of the above cycle resolution and filter functions we can define any of the semantics presenting in Section 2.3. 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 2.3, well-founded semantics resolves positive cycles by assigning false to the propositions, and both even and odd cycles are resolved by assigning $\perp$. So, we use the corresponding cycle resolution functions to define that behavior.

Theorem 20. $(\mathcal{P}_P^-, \mathcal{E}_P^{WF}, \mathcal{L}^{WF}, \mathcal{G})$ *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,
- $f_e(C, M, P) = \{\perp\}$, and
- $f_o(C, M, P) = \{\perp\}$.

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 50, there exists $c \in$ proofmodel(I), with A being the set of all cycles for c, and all functions $\tau : A \to \{\text{true}, \perp\}$ 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 $\tau(a) = \perp$, $\perp \in f(a, C, P)$, 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 $\text{label}(c) = \perp$.

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 $\perp$.

- 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 $f(a, C, P) = \{\perp\}$. So it must be the case that $\tau_I(a) =$ 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 2 we know that all inductive literals must be in the model. From lemma 2 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 $\perp$.

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 chapter that corresponds to that behavior.

Theorem 21. $(\mathcal{P}_P^-, \mathcal{E}_P^{SM}, \mathcal{L}^{SM}, \mathcal{G})$ *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 50, there exists $c \in$ proofmodel(I), with A being the set of all cycles for c, and all functions $\tau : A \to \{\text{true}, \perp\}$ 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 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 2 we know that all inductive literals must be in the model. From lemma 2 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 2 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 2 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 22. $(\mathcal{P}_P, \mathcal{E}_P^{SM}, \mathcal{L}^{SM}, \mathcal{G})$ *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 50, 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 2 we know that all inductive literals must be in the model. From lemma 2 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.

□

CHAPTER 5

GENERALIZATION OF COMPLETION SEMANTICS USING GOAL-DIRECTED, TOP-DOWN ALGORITHM

This chapter presents a generalized goal-directed algorithm for computing partial models. As with previous chapters on generalization, we will be assuming all programs are propositional.

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 (Marple et al., 2012b) 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 83. *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 Chapter 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 $\perp$. 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 (Marple et al., 2012b) and demonstrated in (Marple and Gupta, 2013). 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 "$\langle$" and "$\rangle$" to represent an *internal name*. It is important to note that sample and $\langle sample \rangle$ 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 Chapter 2 is not suitable for practical applications. For the proof-theoretic algorithm we will use the method presented in (Marple et al., 2012b).

To generate the extended program we add rules as follows:

Definition 84. *Let P be some program. Then, for all propositions $p \in props(P)$:*

- *Collect all rules $r \in P$ for which $head(r) = p$.*
- *If no such rules exist add the rule "**not** p." otherwise:*
  - *for each rule r collected and each literal $L \in body(r)$, 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 (Marple et al., 2012b), 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 85. *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 86. *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 87. *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.

Table 5.1: Cycle Resolution Rules for Notable Semantics.

Semantics	**Positive Cycles**	**Even Cycles**	**Odd Cycles**
Fittings 3-Val	SUCCESS($\perp$)	SUCCESS($\perp$) FIX	SUCCESS($\perp$) NOCHK
Well-Founded	FAIL	SUCCESS($\perp$) FIX	SUCCESS($\perp$) 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.

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

```
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
```

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).

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

Definition 88. *Let $\mathcal{R}$ be a cycle resolution rule. The* inverse *of $\mathcal{R}$ is*

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

```
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
```

```
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
```

Definition 89. *D Let $\mathcal{R}$ be a cycle resolution rule.*

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

Definition 90. *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, $f_p(C, M, P) = truthset(\mathcal{R}_p)$,*
  - *if C contains negative literals and $\mathcal{R}_p$ is FIXed, $f_p(C, M, P) = truthset(\mathcal{R}_p)$, and*
  - *if C contains negative literals and $\mathcal{R}_p$ is not FIXed, $f_p(C, M, P) = truthset(\neg\mathcal{R})$.*
- *For some program P, proof model M, and even cycle C, $f_e(C, M, P) = truthset(\mathcal{R}_e)$.*

```
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
```

- *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 91. *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 92. *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 9. *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 $\perp$) or they are the same, but one assigns $\perp$ and the other assigns true. Since $\perp$ 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 93. *Let c_1, c_2 be coinductive proofs. Then c_1* depends *on c_2 if and only if*

- $c_2 \in$ *support*(c_1) *or*
- *there exists $c_3 \in$ 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 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. 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 `FIX`ed 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 9 **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 9, 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 23. *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 $X \subseteq \text{lit}\,(P)$ 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 10. We can take each coinductive proposition and by 10 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 50, there exists $c \in$ proofmodel(M), with B being the set of all cycles for c, and all functions $\tau : B \to \{\text{true}, \bot\}$ 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 $M \not\subseteq M'$.

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. □

CHAPTER 6

THE S(ASP) ALGORITHM

Chapter 2 introduced stable model semantics and the goal-directed method for propositional programs. This chapter discusses an extension of the idea, a predicate goal-directed algorithm.

The work presented in this chapter was equally shared with Dr. Kyle Marple. All theory and design was a group effort. All definitions and proofs are the work of this author. While much of the descriptive text was based off of Dr. Marple's work. The s(ASP) implementation was also the work of Dr. Marple.

6.1 Fundamentals

6.1.1 Extended Stable Models and the s(ASP) Universe

The GL-transform requires propositional programs, and officially, a propositional program can be generated by grounding a predicate program over its Herbrand universe, treating each instance of a predicate as a proposition. ((Cite stable models paper)) However, the Herbrand universe is insufficient for our purposes. One of the defining aspects of s(ASP) is its universe used to ground a program.

It is possible that any member of the Herbrand universe can be constructed and refrenced by the program. s(ASP) requires that there be an infinite number of members in its universe that cannot be constructed or referenced by the unground program, the reasons for which will be discussed later.

Therefore, the s(ASP) algorithm does not strictly implement stable model semantics, but a slight variation we call *extended stable model semantics*. Simply stated, extended stable model sematics is a semantics that grounds over its own universe, which is a superset of the Herbrand universe, and guarentees the above property.

Definition 94. *For some program P, let $\mathcal{H}_P$ be the Herbrand universe for P. Let $\mathcal{P}$ be the set containing all programs, $\mathcal{U} = \{p \mid P \in \mathcal{P}, p \in \mathcal{H}_P\}$, and $\mathcal{U}'$ be the power set of $\mathcal{U}$. Let f be a function that maps a set with cardinality 1 to its element and all other sets to themselves. The s(ASP) universe, denoted by U_s, is $f(\mathcal{U}')$.*

Definition 95. *Let P be a program, and P' be the program generated by grounding P over U_s and treating all instances of a predicate as a proposition. Let M be an interpretation for P'. Then, M is called an* extended stable model *of P if M is the stable model of P'.*

Theorem 24. *Let P be a program, and $\mathcal{H}_P$ be the herbrand universe for P. Then $U_s \setminus \mathcal{H}_P$ has infinite cardinality.*

Proof. The claim holds by definition. (The powerset of a set has higher cardinality) □

Variables and Constraints

The definition for extended stable model semantics requires the program to be ground over U_s, but the s(ASP) algorithm computes partial models without grounding the program. The most obvious step in supporting predicate programs is to accommodate variables. As opposed to negation-as-failure in prolog, the constructive negation employed by s(ASP) relies on extending variables with simple constraints, and thus unification and *disunification* must be modified to accept such variables.

In traditional prolog systems, a variable can either be bound to some other term, allowing to be treat as that term, or unbound, allowing it to be bound to any term. In the s(ASP) algorithm, unbound variables are actually negatively constrained variables, and associated with each variable is a prohibited value list. This list identifies the values the variable cannot be bound to.

If the prohibited value list of a variable `X` contains the constants `a` and `b`, then `X` may be bound to any value *except* `a` and `b`. If the prohibited value list is empty, then the variable is

like a traditional unbound variable and can be bound to any term. It should be noted that a prohibited value list may contain a member that is not ground. However the term must be partially bound. That is it is a compound term that contains a variable.

The s(ASP) algorithm is a goal-directed, top-down algorithm that proves goals in a greedy manner. This means that, operationally, the set of variables and their prohibited value lists may be different at different points in the execution. So, we need an operational representation of variables and other terms.

That being said, except for variables, terms are not affected by the execution. So, refrences to variables and the actual state of the variable will be seperated.

Definition 96. *An* operational term *represents a value during the computation. An operational term can be an operational constant, operational structure, or operational variable. When it is clear that we are talking about an operational term, the word operational may be left off for brevity.*

A structure, $s = f(T_1, T_2, \cdots, T_n)$ *where* T_i*, for* $1 \leq i \leq n$*, is an operational term, has a functor, arity, a list of arguments, and a value.* $\texttt{functor}(s) = f$*,* $\texttt{arity}(s) = n$*, for* $1 \leq i \leq n$*,* $\texttt{arg}_i(s) = T_i$*.*

A constant is a zero-arity structure.

An operational variable*,* V *represents a reference to a variable during computation, and has the following properties:* $\texttt{id}(V)$*, a natural number, uniquely identifies a variable and associates the term with a variable state.*

Definition 97. *Let* T_1, T_2 *be operational terms. Then we say that* T_1 *and* T_2 *are* equal *if*

- *they are both variables with* $\texttt{id}(T_1) = \texttt{id}(T_2)$*, or*
- *they are both structures with* $\texttt{functor}(T_1) = \texttt{functor}(T_2)$*,* $\texttt{arity}(T_1) = \texttt{arity}(T_2)$*, and for all* $1 \leq i \leq \texttt{arity}(T_1)$*,* $\texttt{arg}_i(T_1) = \texttt{arg}_i(T_2)$*.*

As stated above, an operational variable is simply a reference to a variable, but does not maintain the state of the variable. This information will be kept separately by variable states.

Definition 98. *A* variable state *represents the state of an operational variable during execution. A variable state, V, has the following properties:*

- `id`(V) *a natural number to uniquely identify the variable.*
- `type`(V) *is the type of variable, and can be* bound, unbound, *or* loop. *A loop variable is a special unbound variable that should only appear in the chs. Normally in the chs a variable is considered universal, but loop variables may be ground with any subset of its domain, allowing for an infinite number of groundings(and therefore an infinite number of partial answer sets). Loop variables will be presented in more details in 6.2.2.*
- `binding`(V) *is an operational term this variable behaves as. If V is not of type* bound, *the* `binding`(V) *is an operational variable with the same* `id` *as V.*
- `prohibited`(V) *is a set of values that V cannot unify with. When V is* unbound *this set is directly associated with the variable. When V is of type* bound, *if* `binding`(V) *is a variable then* `prohibited`(V) = `prohibited(binding(`V`))`, *and is* undetermined *otherwise.*

A variable state set *is a set of variable states such that for any two elements V_1 and V_2,* `id`$(V_1) =$ `id`$(V_2) \Rightarrow V_1 = V_2$. *For some variable state set S and $V \in S$, we say $S[$*`id`$(V)] = V$.

For convenience, there should be some shortcuts for referring to variable properties. First, we would like to modify the state without explicitly stating what part of the state did not change.

Definition 99. *Let V be a variable state. For some property of variable state,* `prop`*, and X in the codomain of* `prop`*. We say* $\texttt{prop} = X$ *is true for V if* $\texttt{prop}(V) = X$*.*

Now, let E be a list of such equalities. $V[E]$ is a variable state for with each equality in E is true, and all properties without an equality in E is the same as V.

Secondly, we would like a simple notation that represents traversing variable bindings to treat variables as what they are bound to. This will be called the value of the term.

Definition 100. *Let ψ be some operational state and T be some operational term. We can define the function* $\texttt{value}_\psi$ *as follows:*

- *if* $\texttt{type}(V) = \texttt{bound}$*, where* $V = \texttt{vars}(\psi)[\texttt{id}(T)]$*, and* $\texttt{value}_\psi(\texttt{binding}(V))$ *is not undefined then* $\texttt{value}_\psi(T) = \texttt{value}_\psi(\texttt{binding}(V))$
- *if T is a structure, and we can construct a structure T' with the same functor and arity and for all* $1 \leq i \leq |\texttt{arity}(T)|$*,* $\texttt{arg}_i(T') = \texttt{value}_\psi(\texttt{arg}_i(T))$*, then* $\texttt{value}_\psi(T) = T'$*.*
- *in all other cases* $value_\psi(T)$ *is undefined.*

We say T is ground *with respect to ψ if and only if* $\texttt{value}_\psi(T)$ *is not undefined.*

6.1.2 Operational State, Rules, and Transformations

Operational terms and variable states can be used to keep track of changes in terms during execution as the domain of variables change. Before defining the various parts of the s(ASP) algorithm, a framework must be presented to allow the tracking and manipulation of the state pf execution. The most obvious starting point is to define what such a state is.

Definition 101. *An* operational state*, ψ, is a variable state set (*$\texttt{vars}(\psi)$*), set of Goals (*$\texttt{chs}(\psi)$*), and a sequence of goals(*$\texttt{callstack}(\psi)$*).*

Modifying and referencing things in the state can be cumbersome due to the depth things are nested. We will define some notation to make the process more convenient.

Definition 102. *Let Φ be an operator such that when operated on an operational state it results in a new operational state. Then, Φ is called a* state transformation.

Definition 103. *Let V be a variable state. For some $n > 0$ and for all $1 \leq i \leq n$, let $\texttt{prop}_i$ be some property and X_i some value that can be assigned to the property. Then, $V[\texttt{prop}_1 = X_1, \texttt{prop}_2 = X_2, \cdots, \texttt{prop}_n = X_n]$ is the variables state, V' such that for all $1 \leq i \leq n$, $\texttt{prop}_i(V') = X_i$ and for all other properties, $\texttt{prop}$, $\texttt{prop}(V') = \texttt{prop}(V)$.*

Let ψ be an operational state. Then when used as a state transformation, $V[\texttt{prop}_1 = X_1, \texttt{prop}_2 = X_2, \cdots, \texttt{prop}_n = X_n]$ generates a new operational state ψ' such that $\texttt{chs}(\psi') = \texttt{chs}(\psi)$, $\texttt{callstack}(\psi') = \texttt{callstack}(\psi)$, and $\texttt{vars}(\psi') = (\texttt{vars}(\psi) \setminus \{V\}) \cup \{V'\}$.

Definition 104. *Let ψ be an operational state and g be some goal. Then $g \rightarrow \texttt{chs}$ is an state transformation such that when applied to ψ is generates a new state, ψ' where $\texttt{vars}(\psi') = \texttt{vars}(\psi)$, $\texttt{callstack}(\psi') = \texttt{callstack}(\psi)$, and $\texttt{chs}(\psi') = \texttt{chs}(\psi) \cup \{g\}$.*

Definition 105. *Let $apply(X, \psi)$ be the result of applying state transformation X to operation state ψ. Let ψ be some state, $n > 0$, and for all $1 \leq i \leq n$, X_i be some state transformation. Let $\gamma_0 = \psi$, and for all $1 \leq i \leq n$, $\gamma_i = apply(X_i, \gamma_{i-1})$. Then, $\psi\langle X_1.X_2, \cdots, X_n\rangle = \gamma_n$.*

Next, a simple notation can be defined that represents traversing variable bindings to treat variables as what they are bound to. This will be called the value of the term.

Definition 106. *Let ψ be some operational state and T be some operational term. We can define the function $\texttt{value}_\psi$ as follows:*

- *if $\texttt{type}(V) = \texttt{bound}$, where $V = \texttt{vars}(\psi)[\texttt{id}(T)]$, and $\texttt{value}_\psi(\texttt{binding}(V))$ is not undefined then $\texttt{value}_\psi(T) = \texttt{value}_\psi(\texttt{binding}(V))$*
- *if T is a structure, and we can construct a structure T' with the same functor and arity and for all $1 \leq i \leq |\texttt{arity}(T)|$, $\texttt{arg}_i(T') = \texttt{value}_\psi(\texttt{arg}_i(T))$, then $\texttt{value}_\psi(T) = T'$.*

- *in all other cases* $value_\psi(T)$ *is undefined.*

We say T *is* ground *with respect to* ψ *if and only if* $\texttt{value}_\psi(T)$ *is not undefined.*

A simple notation to define whether or not a specific variable exists somewhere in a state or term will be very useful when talking about variables. Since variables are always within some context.

Definition 107. *Let* ψ *be an operational state, and let* X *be an operational variable. If there exists* $V \in \texttt{vars}(\psi)$ *such that* $\texttt{id}(V) = (id)(X)$ *then* X *is said to be* in ψ*. Let* T *be an operational term. If* T *is a variable and* $T = X$ *then* X *is* in T*. If* T *is a structure with arity greater than zero and there exists* $1 \leq i \leq |\texttt{arity}(T)|$ *such that* X *is in* $\texttt{arg}_i(T)$*, then* X *is* in T*.*

Lastly, there are cases when it would be convenient to make a copy of a term at a specific point in time without the term being affected by further changes in the state of the original term. Such a copy will be referred to as a clean copy.

Definition 108. *Let* ψ *be an operational state, and* T *some operational term. Let* V *be the set of operational variables in* T*, and* $\mathcal{V} \subseteq V$*. Let* X *be a variable state set such that* $|X| = |V|$*,* $X \setminus \texttt{vars}(\psi) = X$*, and* $\texttt{vars}(\psi) \cup X$ *is a variable state set. Let* f *be a function that maps an element in* V *to an element in* X*.*

Let scrub be a function that maps an operational term, T_1*, to another operational term,* T_2*, in the following manner:*

- *if* T_1 *is an operational variable, and is not* `bound` *then* T_2 *is an operational variable such that* $\texttt{id}(T_2) = \texttt{id}(f(T_1))$*,*

- *if* T_1 *is an operational variable, and is* `bound` *then* $T_2 = scrub(\texttt{binding}(T_1))$*, and*

- *otherwise, T_1 must be an operational structure, meaning T_2 is an operational structure with* $\texttt{functor}(T_1) = \texttt{functor}(T_2)$*,* $\texttt{arity}(T_1) = \texttt{arity}(T_2)$*, and for all* $1 \leq i \leq \texttt{arity}(T_1) : \texttt{arg}_i(T_2) = \mathit{scrub}(\texttt{arg}_i(T_1))$*.*

We can compute a clean copy *of T with respect to ψ:* $\texttt{clean}(T, \psi) = (\mathit{scrub}(T), \psi')$*, where ψ' is the operational state generated by replacing* $\texttt{vars}(\psi)$ *with* $\texttt{vars}(\psi) \cup X$*.*

Alternatively, for goals that should have loop variables, $\texttt{cloop}(\mathcal{V}, T, \psi) = (\mathit{scrub}(T), \psi')$*, where ψ' is the operational state generated by replacing* $\texttt{vars}(\psi)$ *with* $\texttt{vars}(\psi) \cup X'$ *and X' is the result of replacing all $f(v)$ such that $v \in \mathcal{V}$ with a new variable state that is produce by changing* `type` *to* `loop`*..*

With a way to track the state of execution, there must connect the program execution with the program itself. This will be done by differentiating between the rules (clauses) of the program and stateful rules generated from them, used during execution.

Definition 109. *A* body of goals *is represented by a strict totally ordered set of goals. The relation "$<$" is used to describe the total order. In addition, for some body of goals G:*

- *the* minimal goal *is a goal $g \in G$ such that for all $g' \in G$, with $g \neq g'$, $g < g'$.*
- *the* maximal goal *is a goal $g \in G$ such that for all $g' \in G$, with $g \neq g'$, $g' < g$.*
- *for some $g \in G$, there exists a body of goals, G', called the* top *such that $g' \in G' \iff g' \in G \wedge g < g'$.*
- *for some $g \in G$, there exists a body of goals, G', called the* bottom *such that $g' \in G' \iff g' \in G \wedge g' < g$.*

Definition 110. *An* operational rule *r is a goal,* $\texttt{head}(r)$*, and a body of goals,* $\texttt{body}(r)$*.*

With the ability to represent the program during execution, we must now be able to convert between the representation. Informally, each operational rule, goal and term is associated with a rule, goal or term in the program. It is called an instance of that rule, goal or term.

Definition 111. *Let P be a program, ψ be an operational state, and $r \in P$. Then the* context *of r, $\chi_{r,\psi}$ is a function that maps variables to variable states in ψ.*

Definition 112. *Let ψ be an operational state, T be a term, and χ_ψ be a context for T. Then the operational term T' is an instance of T with respect to ψ if*

- *when T is a variable, T' is an operational variable with $\texttt{id}(T') = \chi_\psi(T)$, and*
- *when T is a structure (or constant), T' is an operational structure with the same functor and arity as T and for all $1 \leq i \leq |\texttt{arity}(T')|$ $\texttt{arg}_i(T')$ is an instance of the corresponding argument of T.*

Let B be a conjunction of structures. Let B have a total ordering. Then, the body of goals G is an instance of B if $g \in G$ if an only if it is an instance of some $T \in B$, and for $g_1, g_2 \in G$, $g_1 < g_2$ if the term g_1 is an instance of comes before the term g_2 is an instance of in B.

Definition 113. *Let P be a program, ψ an operational state, and $r \in P$ be a rule. Let χ_{r,ψ_2} be the context for r, where ψ_2 is constructed from ψ. Then the operational rule r' is an instance of r if*

- $\texttt{head}(r')$ *is an instance of* $\texttt{head}(r)$ *with respect to ψ_2, and*
- $\texttt{body}(r')$ *is an instance of* $\texttt{body}(r)$.

The s(ASP) method is divided into a collection of operations that are tightly interconnected. These operations will be represented as a function that modifies the current execution state.

Definition 114. *An* operational transformation *is a function that maps a body of goals and an operational state to a strict totally ordered set of pairs consisting of a body of goals and an operational state.*

6.1.3 Constructive Unification and Disunification

Now that we have introduced negatively constrained variables, unification and disunification must be extended to work with them. To differentiate the modified versions from the originals, we will refer to them as *constructive* unification and disunification. For cases where neither argument contains a negatively constrained variable, the constructive algorithms are identical to the traditional ones.

Constructive disunification is the dual of constructive unification with one exception: in accordance with the restrictions given in Section 6.1.4, constructive disunification of two negatively constrained variables will produce an error. The remaining cases are as follows:

- Constructive disunification of a negatively constrained variable and a non-variable value will always succeed, adding the non-variable value to the variable's prohibited value list.

- Constructive disunification of two compound terms is performed by first testing functors and arities. If either of these does not match, the operation succeeds deterministically. Otherwise, the pairs of corresponding arguments are handled recursively. *Non-deterministic* success occurs as soon as the operation succeeds for a pair of arguments, with subsequent pairs tested upon backtracking.

- In cases where neither argument contains a negatively constrained variable, the result is identical to that of traditional disunification.

Given a variable `X` whose prohibited value list contains `a`, disunifying `X` with a constant `c`, i.e. solving `X \= c` where `\=` represents the disunification operator, will extend the

prohibited value list of `X` to `[a,c]`, i.e. `X` cannot be bound to either `a` or `c`. Under our program restrictions, discussed further in Section 6.1.4, the disunification of two negatively constrained variables is considered illegal. There is, however, an exception to this rule involving variables in even loops, discussed in Section 6.2.2.

More formally, we must construct an operational transformation.

Definition 115. *Let T_1 and T_2 both be operational terms. Let Q be a body of goals and ψ be an operational state. Then the constructive disunification of T_1 and T_2, represented by $T_1 \backslash\!= T_2$, defines an operational transformation $\delta^d_{T_1,T_2}$ as follows:*

If T_1 and T_2 are equal, $\delta^d_{T_1,T_2}(Q,\psi) = \emptyset$.

If T_1 and T_2 are both structures *and not equal, then*

- *if* $\texttt{functor}(T_1) \neq \texttt{functor}(T_2)$ *or* $\texttt{arity}(T_1) \neq \texttt{arity}(T_2)$, $\delta^d_{T_1,T_2}(Q,\psi) = \{(Q,\psi)\}$,
- *otherwise,* $\delta^d_{T_1,T_2}(Q,\psi) = \{(Q,\psi') \mid 1 \leq i \leq |\texttt{arity}(T_1)|, (Q,\psi') \in \delta^d_{\texttt{arg}_i(T_1),\texttt{arg}_i(T_2)}(Q,\psi)\}$

Suppose that either T_1 or T_2 is a variable, and T_1 is not equal to T_2. Without loss of generality, assume T_1 is a variable. Let $V_1 = \texttt{vars}(\psi)[\texttt{id}(T_1)$. If $\texttt{type}(V_1)$=bound, *then* $\delta^d_{T_1,T_2}(Q,\psi) = \delta^d_{\texttt{binding}(V_1),T_2}(Q,\psi)$. *If* $\texttt{type}(V_1)$ *is* unbound *then*

If $\texttt{value}_\psi(T_2)$ is a structure: *Let $V_2 = V_1[\texttt{prohibited} = \texttt{prohibited}(V_1) \cup \{\texttt{value}_\psi(T_2)\}]$.*
Let $\psi' = \psi[\texttt{vars} = (\texttt{vars}(\psi) \setminus \{V_1\}) \cup \{V_2\}]$. Then, $\delta^d_{T_1,T_2}(Q,\psi) = \{(Q,\psi')\}$.

If $\texttt{value}_\psi(T_2)$ is a variable: *$\delta^d_{T_1,T_2}(Q,\psi)$ is not defined. In Section 6.1.4, we assume that such a situation does not occur and explain why we make such an assumption.*

As can be seen from the above definition, constructive disunification of structures can be non-determinisitic. For instance, consider a(X, Y)\=a(1, 2). If $X \backslash\!= 1$ is true, then Y can remain unchanged. On the other hand, X can be left the same and $Y \backslash\!= 2$ must be made true.

Now, consider unification. The cases for **constructive unification** are as follows:

- Constructive unification of a negatively constrained variable with a non-variable value will succeed if the non-variable value does not constructively unify with any element in the variable's prohibited value list.

- Constructive unification of two negatively constrained variables will always succeed, setting their shared prohibited value list to the union of their original lists.

- Constructive unification of two compound terms is performed recursively: first, the functors and arities are tested, then each pair of corresponding arguments is constructively unified.

- In cases where neither argument contains a negatively constrained variable, the result is identical to that of traditional unification.

If two variables, X and Y were to be unified, the prohibited value list for X contain just the constant a, and the prohibited value list for Y contained just the constant b, then as a result of that unification the prohibited value list for both variables will be changed to contain both a and b. Afterwards, unifing either variable with a or b will always fail.

If instead of two variables one side is a structure, then the structure must be disunified with everything in the prohibited value list. So, consider X\=s(a),X=s(Y). After the execution of these two goals, the prohibited value list for the variable Y must contain a.

Definition 116. *Let $\mathbb{F}$ be a function that maps a operational transformation, a body of goals, and a set of operational states to a set of operational states as follows:*

$$\mathbb{F}(\delta, Q, S) = \{\psi' \mid \psi \in S, (Q, \psi') \in \delta(Q, \psi)\}.$$

Definition 117. *Let T_1 and T_2 both be operational terms. Let Q be a body of goals and ψ be an operational state. Then the constructive unification of T_1 and T_2, represented by $T_1 == T_2$, defines an operational transformation $\delta^u_{T_1,T_2}$ as follows:*

Suppose, T_1 *and* T_2 *are both* structures. *If* $\texttt{functor}(T_1) = \texttt{functor}(T_2)$ *and* $\texttt{arity}(T_1) = \texttt{arity}(T_2)$, *with* $\psi_0 = \{\psi\}$, *and for every* $1 \leq i \leq \texttt{arity}(T_1)$, $\psi_i = \mathbb{F}(\delta^u_{\texttt{arg}_i(T_1),\texttt{arg}_i(T_2)}, Q, \psi_{i-1})$ *where* $\psi_{\texttt{arity}(T_1)} \neq \emptyset$, $\delta^u_{T_1,T_2}(Q,\psi) = \{(Q,\psi') \mid \psi' \in \psi_{\texttt{arity}(T_1)}\}$. *Otherwise* $\delta^u_{T_1,T_2}(Q,\psi) = \emptyset$

Suppose that either T_1 *or* T_2 *is a variable. Without loss of generality, assume it is* T_1, *and let* $V_1 = \texttt{vars}(\psi)[\texttt{id}(T_1)]$ *(If* T_1 *is not in* ψ *behavior is undefined). If* $\texttt{type}(V_1)$ *is* bound, *then* $\delta^u_{T_1,T_2}(Q,\psi) = \delta^u_{T_3,T_2}(Q,\psi)$, *where* $T_3 = \texttt{binding}(V_1)$. *If* V_1 *is* `unbound` *or* `loop`, *there are two cases:*

$\texttt{value}(T_2)$ is a structure. *Let* ω *be a one to one function, maping a natural number greater than zero and less than or equal to* $|\texttt{prohibited}(V_1)|$ *to a member of* $\texttt{prohibited}(V_1)$. *Let* $\psi_0 = \{\psi\}$ *and for all* $1 \leq i \leq |\texttt{prohibited}(V_1)|$, $\psi_i = \mathbb{F}(\delta^d_{\omega(i),T_2}, Q, \psi_{i-1})$. *Let* T_3 *be a variable state with* $\texttt{id}(T_3) = \texttt{id}(T_1)$, $\texttt{type}(T_3) = \texttt{bound}$, *and* $\texttt{binding}(T_3) = T_2$. *Then,* $\delta^u_{T_1,T_2}(Q,\psi) = \{(Q,\psi'') \mid 0 < i \leq |\texttt{prohibited}(V_1)|, (Q,\psi') \in \psi_i, \texttt{callstack}(\psi'') = \texttt{callstack}(\psi'), \texttt{chs}(\psi'') = \texttt{chs}(\psi'), \texttt{vars}(\psi'') = \{T_3\} \cup (\texttt{vars}(\psi') \setminus \{V_1\})\}$.

T_2 is a variable. *Let* $V_2 = \texttt{vars}(\psi)[\texttt{id}(T_2)]$. *If* $\texttt{type}(V_2) = \texttt{bound}$ *then* $\delta^u_{T_1.T_2}(Q,\psi) = \delta^u_{T_1,T_3}(Q,\psi)$ *where* $T_3 = \texttt{binding}(V_2)$.

Otherwise let V_3 *be a variable state such that* $\texttt{id}(V_3) = \texttt{id}(T_1)$, $\texttt{type}(V_3) = \texttt{unbound}$, *and* $\texttt{prohibited}(V_3) = \texttt{prohibited}(V_1) \cup \texttt{prohibited}(V_2)$. *Let* V_4 *be a variable state such that* $\texttt{id}(V_4) = \texttt{id}(T_2)$, $\texttt{type}(V_4) = \texttt{bound}$, *and* $\texttt{binding}(V_4) = T_1$. *Let* ψ' *be an operational state such that* $\texttt{chs}(\psi') = \texttt{chs}(\psi)$, $\texttt{callstack}(\psi') = \texttt{callstack}(\psi)$, *and* $\texttt{vars}(\psi') = \{V_3, V_4\} \cup (\texttt{vars}(\psi) \setminus \{V_1, V_2\})$. *Then,* $\delta^u_{T_1,T_2}(Q,\psi) = \{(Q,\psi')\}$.

In all other cases $\delta^u_{T_1,T_2}(Q,\psi) = \emptyset$.

Constructive Negation and Dual Rules

In Section 2.2, the completion of a program was defined for propositional programs. In that section, the completion was simulated with dual rules, forming an extended program.

This must be extended to work with non-propositional programs. In fact, the dual rules are needed for one of cornerstones of the s(ASP) algorithm: constructive negation.

Traditionally, in program systems, negation-as-failure is implemented by using SLD resolution to prove the atom, and inverting success and failure. There are a couple problems s(ASP) faces with this. First of all, it is not enough to know that an atom fails, but the algorithm must know *why* it failed. This is necessary for constraints due to odd cycles. Secondly, Since any variables in the atom are treated as existential during execution, by negating the result the variables are treated as universal in the negation. This does not match how the program should be grounded.

In Section 2.7, a less computationally intensive method of generating dual rules was presented. This method makes use of intermediate utility propositions to get the same effect.

For example, given a proposition `p` with the clauses

```
p :- a, not b.
p :- r.
```

the dual of `p` would be

```
not p :- np1, np2.
np1 :- not a.
np1 :- b.
np2 :- not r.
```

The above method works for propositional programs, but it runs into few problems when used in presence of variables. The first problem has do do with the interaction of literals in the body. Through unification and disunification a literal could modify the state of a variable, and the success of another literal may depend on that modification. For instance, consider the rule p(X):−q(X),r(X).. The above method would produce the following dual:

```
not p(X) :- not q(X).
not p(X) :- not r(X).
```

Mathematically, it should make no difference, but operationally q(X) could change the state of X, modifying the behavior of r(X). An example of this would be comparison. In this chapter and the next, it will be assumed that comparisons are facts with an infinite number of rules. The math will work out fine, and the definitions and proofs do not need to take things like arithmetic into account. Operationally, though, each comparison must be ground at the time of execution. So if q(X) bound X to some number and $r(X)$ had a comparison it can be seen that the second rule of the dual would not work, since there would be a comparison that is not ground.

To remedy this problem, each dual rule must include every literal before the negated one. So, in the previous example the dual would look like this:

```
not p(X) :- not q(X).
not p(X) :- q(X), not r(X).
```

Another problem comes from unification with a rule's head. Since the s(ASP) algorithm is based off the galliwasp algorithm, which is in turn based of the SLD resolution, it is clear that during execution, a literal would be used to select a rule, whose body would replace that literal. But, when a rule is select, it is not enough to replace the literal with the body, but first, the literal and the head must be unified. It is easy to forget this fact, but the dual rules must account for it. The new dual rule algorithm accounts for it by internally rewriting the rule to make the unification explicit. When computing the dual of a rule the head is analysed. If a variable appears more than once in the head each occurrence after the first is replaced with a new variable and the new variables are unified with the old variable in beginning of the body. If an argument is not a variable, then it is replaced with a new variable and unified with that variable in the beginning of the body. As an example consider the rule t(A,A).. Before generating the dual it is internally rewritten to t(A,B) :- A = B., which would generate the dual **not** t(A,B) :- A\=B..

The last problem has to do with body variables. A body variable is a variable that appears in the body of the rule, but not in the head. Variables in the head have an implicit

universal quantifier. This quantifier is universal by definition. Body variables have an implicit existential quantifier. That is, only one value for that variable that makes the body true is needed for the head to be true. This quantifier must be proved. Consider the following:

q(X) :- **not** p(X,Y).

is equivalent to

$$\forall X(q(X) \leftarrow \exists Y \neg p(X,Y))$$

Duals negate the body, and by extension must negate the quantifiers in the body as well. So the dual of the above formula would look like this:

$$\begin{aligned} &\forall X(\neg(q(X)) \leftarrow \neg(\exists Y \neg p(X,Y))) \\ \equiv &\forall X(\neg q(X) \leftarrow \forall Y \neg(\neg p(X,Y))) \\ \equiv &\forall X(\neg q(X) \leftarrow \forall Y p(X,Y)) \end{aligned}$$

As this example illustrates, the dual of an existential quantifier is an universal quantifier. Since the existential quantifier in the original rule must be proved for each instance of the head, the universal quantifier of the dual must be proved for each instance of its head. To generate such duals, a special for-all mechanism has been created that relies on negatively constrained variables. This is a special builtin goal of the form forall (V,G) where V is an unbound variable with an empty prohibited value list and G is some goal, and it can only be included by the system.

When generating duals of rules that have body variables the for-all is used. First, the generation proceeds as usual as if the body variables are not present. The head of the top level rule is renamed to some internal utility name, and the body variables are added to the head, increasing the head's arity. Finally a new rule with the original duals head is generated with a for-all for the body variable with the new utility literal as the goal. If there is more

than one body variable the for-alls are nested with each outer for-all using a forall as the goal.

So, continuing with the above example the dual rule would look like this:

```
not q(X) :- forall(Y, nq1(X,Y)).
nq1(X,Y) :- p(X,Y).
```

The for-all mechanism is formally defined in Section 6.2 as part of the "step" operational transformation. Informally, at runtime, a for-all is executed by executing G. If G succeds, but V was bound then a fail is forced and G's execution is backtracked. If V is unbound, then G has been proven for every value in the s(ASP) universe except the one listed in the prohibited value list. So, for each value in the list, Y is bound to it and G is executed once more. If G succeeds for all values in the prohibited value list then the for-all succeeds. If G fails for one such value, or if G fails to succeed with V unbound then the for-all fails.

Consider the following program with dual rules:

```
p :- not q(X).
q(Y) :- Y = a.
q(Y) :- Y \= a.

not p :- forall(X, np1(X)).
np1(X) :- q(X).

not q(Y) :- nq1(Y), nq2(Y).
nq1(Y) :- Y \= a.
nq2(Y) :- Y = a.
```

and query **not** p. First the rule for **not** p fill execute a for-all, which will execute np1(X), followed by q(X). The first rule for q(X) will bind X to a. Since this is in a for-all and X is the for-all's variable, the rule will fail instead of succeeding. The second rule will add a to X's prohibited value list and succeed. Since X is not bound, the np1(a) will then be executed. Again, q(a) will be executed, which succeds with the first rule. So, forall (X, np1(X)) succeeds and by extension **not** p.

As stated earlier, the forall literal can only be generated by the system, and is not exposed to the user. This is because the for-all variable must be unbound with an empty prohibited list. If this property is not fulfilled the for-all will not work as is. Consider a single rule "p(1)." and a for-all " forall (X,p(X))". The for-all will fail, but this is correct when dealing with the s(ASP) universe. There are an infinite number of values for which p(X) is false. However if the Herbrand universe was used, then the for-all should succeed since X can only take on the value 1.

It is also important to know that the dual rule algorithm can produce undesirable results. In Section 6.1.4, a collection of runtime assumption/restrictions are given. The s(ASP) system only behaves properly when they are satisfied. One such assumption is that two unbound variables will never be disunified. However, the dual rule algorithm can generate such situations automatically without anything the user can do to stop it. Since the head of a rule is unified with the literal being proven, it's dual will generate a disunification. For example,

```
p([X | T]) :- q(X), p(T).
```

The first step of the dual rule algorithm is to move the list to the body. But this makes the variables X and T body variables. This means for-alls are needed.

```
not p(Z) :- forall(X, forall(T, np1(Z, X, T))).
np1(Z, X, T) :- Z \= [X | T].
np1(Z, X, T) :- Z = [X | T], not q(X).
np1(Z, X, T) :- Z = [X | T], q(X), not p(T).
```

Consider the query **not** p([A|B]). The first rule for np1 will result in [A | B]\=[X | T]. Recalling the algorithm for disunification, there are two possibilities: A\=X and B\=T. Both of which violate the restriction. The same situation arises when a variable occurs multiole times in the head. This is because, after the first occurrence, each instance is replaced with a new variable and the two variables are unified in the body. This means they will be disunified in the dual. Consider the following rule and its dual:

```
t(A,A).
not t(A,B) :- A \= B.
```

Even though, the original rule is perfectly fine to call with both arguments as unbound variables. The same cannot be said for the dual. This is because two unbound variables will be disunified. Once again violating the restriction. This will likely be a problem until the restrictions themselves can be lifted.

6.1.4 Runtime Restrictions

The s(ASP) algorithm is a new approach to stable models. As such there are still much work needed. While the goal of the project was to support as many programs as possible some restrictions are still necessary with the current algorithm. The following restrictions will be assumed to be satisfied when defining and proving behavior.

- All comparisons and arithmetic is ground at the time they are executed.
- A unbound variable cannot be disunified with another unbound variable.

The first restriction is a fairly standard assumption. Since operators are trated as syntactic sugar for some theoretical predicate that defines the same behavior, this restrictions will not affect the theory.

The second restriction has a greater affect on the system. As seen in the previous section, it is difficult for the user to guarantee that this restriction is satisfied. It may be possible to lift this restriction by implementing a simple constraint solver, but such a solver would not be efficient.

6.2 Advanced Mechanisms

6.2.1 Modified CoSLD Resolution

The s(ASP) algorithm is based on the ground method described in Section 2.7, and like it s(ASP) uses a modified coSLD resolution. The modified coSLD resolution must be modified further to account for negativly constrained variables, but simply using unification to detect cycles – as is done in coinductive logic programming – will not work in this case. This section presents the modifications necessary.

Under the method for propositional programs, coinductive failure occurs when the negation of a goal is present in the CHS (Marple et al., 2012a). For example, if **not** p is found in the CHS when checking p, the call to p will fail. When dealing with propositional programs, it is sufficient to simply fail when a match for the negation is found. The most obvious extension for working with non-ground programs is to unify the negation with things in the CHS. However, this can lead to incorrect behavior when combined with CHS entries that include negatively constrained variables. Consider a program consisting of the following rule, with its dual added for convenience:

```
pi(X) :- X = 3.14.
not pi(X) :- X \= 3.14.
```

Correct behavior requires that **not** pi(X) should always succeed with X = 3.14. However, this may not happen if we rely on ordinary unification to check for coinductive failure. For instance, if we execute the program with the query pi(Y), **not** pi(X), the goal pi(Y) will succeed for Y = 3.14 and execution will move on to **not** pi(X). However, the negation of **not** pi(X) will unify with the CHS entry for pi(Y), causing the query to incorrectly fail.

The solution s(ASP) uses to avoid the problem once again relies on constructive negation. Instead of simply failing when the negation of a goal unifies with an entry in the CHS, coinductive failure is viewed as a filter, and allows bindings whose negation does not unify with a CHS entry to succeed. This is accomplished by constraining variables in the call such

that the call's negation will no longer unify with any entry in the CHS. In the above example, when **not** pi(X) is tested, X will be constrained against 3.14, preventing its negation, pi(X), from unifying with the CHS entry for pi(3.14).

There are two things worth noting. First, the *negation* of a goal is checked rather than its *dual*. Recall that a dual succeeds in anf only if the original fails. A negation is simply adding or removing the *not* to/from the literal. For instance, the dual of pi(3.14) is **not** pi(X) where $X \backslash = 3.14$, but the negation of pi(3.14) is simply **not** pi(3.14).

The second thing of note is that the coinductive failure check may be non-deterministic. Consider the following rule:

```
q(X) :- X \= 2, X \= 3.
```

Given the query q(X), **not** q(Y), at the time **not** q(Y) is called, the CHS will contain q(X), with X constrained against both 2 and 3. This leaves two ways for the coinductive failure check to succeed: Y may be set to either 2 or 3. This choice will be made non-deterministically: 2 will be selected first, but 3 may be chosen when backtracking.

In fact, non-determinism may also arise when testing goals with more than one argument. Consider a modification of the previous rule:

```
q(X, Y) :- X \= 2, Y \= 3.
```

Now, given the query q(X, Y), **not** q(A, B), at the time **not** q(A, B) is called, the CHS will contain q(X, Y), with $X = 2$ and $Y = 3$. Once again this leaves two ways for the coinductive failure check to succeed: A may be set to 2 or B may be set to 3.

To ensure that all cases are considered when executing the coinductive failure check, each argument is considered separately first to last, with subsequent arguments being selected on backtracking once all choices for the previous argument have been exhausted. Consider the rules:

```
q(W, X) :- W \= 2, W \= 3, X \= 2, X \= 3.
q(Y, 3) :- Y \= 2, Y \= 3.
```

Consider the query q(W, X), q(Y, 3), **not** q(A, B). At the time **not** q(A, B) is called the CHS will contain both q(W, X), with W = 2, W = 3, X = 2, X = 3 and q(Y, 3), with Y = 2, Y = 3. When executing the coinductive failure check for **not** q(A, B), the first argument will be examined first, and the goal will be allowed to succeed first with A = 2 and then with A = 3. With all options exhausted for the first argument , further backtracking will lead to the second argument being examined. At this point, B = 2 is the only valid option. Therfore, the coinductive failure check for **not** q(A, B) will succeed up to three times: once each for A = 2, A = 3 and B = 2. Together, these cases cover all possible scenarios where **not** q(A, B) does not unify with any element of the CHS.

So, this form of coinductive failure can be viewed as two parts: a transformation that converts a goal to several goals that do not unify with some goal in the chs, and a transformation that will apply this over all elements of the chs.

Definition 118. *Let ψ be an operational state, and T_1 and T_2 be operational terms. Let Q be a body of goals. Then the constructive mismatch of T_1 and T_2 defines an operational transformation $\delta^m_{T_1,T_2}$ as follows:*

If T_1 and T_2 are equal, $\delta^m_{T_1,T_2}(Q,\psi) = \emptyset$.

If T_1 and T_2 are both structures *and not equal, then*

- *if* $\texttt{functor}(T_1) \neq \texttt{functor}(T_2)$ *or* $\texttt{arity}(T_1) \neq \texttt{arity}(T_2)$, $\delta^m_{T_1,T_2}(Q,\psi) = \{(Q,\psi)\}$,
- *otherwise,* $\delta^m_{T_1,T_2}(Q,\psi) = \{(Q,\psi') \mid 1 \leq i \leq |\texttt{arity}(T_1)|, (Q,\psi') \in \delta^m_{\texttt{arg}_i(T_1),\texttt{arg}_i(T_2)}(Q,\psi)\}$

If T_1 is a structure and T_2 an unbound variable, then let V be a variable state set such that $|V| = \texttt{arity}(T_1)$, $V \cap \texttt{vars}(\psi) = \emptyset$, and $V \cup \texttt{vars}(\psi)$ is a variable state set. Let ψ_1 be the state resulting from putting T_1 in T_2's prohibited list. Let T_3 be a structure with the same functor and arity as T_1, but each argument is a different variable from V, let ψ_2 be the result of putting the variable states in V into ψ's variable state set, and unifying T_2 and T_3. Then, $\delta^m_{T_1,T_2}(Q,\psi) = \{(Q,\psi_1)\} \cup \delta^m_{T_1,T_3}(Q,\psi_2)$.

Suppose that T_1 is a variable, and is not equal to T_2. Let $V_1 = \texttt{vars}(\psi)[\texttt{id}(T_1)]$. Note that when adding to the chs, any bound variables will be replaced with their value. So only the case of unbound and loop variables need to be considered. If $\texttt{type}(V_1)$ *is* unbound *then*

if $\texttt{prohibited}(V_1) = \emptyset$**:** $\delta^m_{T_1,T_2}(Q,\psi) = \emptyset$*, or*

If $\texttt{prohibited}(V_1) \neq \emptyset$**:** $\delta^m_{T_1,T_2}(Q,\psi) = \{X \mid T_3 \in \texttt{prohibited}(V_1), (T_4,\psi') = \texttt{clean}(T_3,\psi), X \in \delta^u_{T_4,T_2}(Q,\psi')\}$.

Finally, if $\texttt{type}(V_1) = \texttt{loop}$ *then* $\delta^m_{T_1,T_2}(Q,\psi) = \delta^d_{T_1,T_2}(Q,\psi)$.

Unlike disunification, the mismatch operation is not commutative. The terms in the chs behave differently than goals that are being proven. That is, except for the case of loop variables. They are an exception to the rule that the state of variables in the chs cannot change. With the mismatch operation, the coniductive failure check can be defined.

Definition 119. *Let ψ be an operational state, T a goal, and Q some body of goals. We can define the operational transformation δ^f_T as follows: Let ω be a function that maps an integer $1 \leq i \leq |\texttt{chs}(\psi)|$ to an element of $\texttt{chs}(\psi)$. With $\psi_0 = \{\psi\}$, for all $1 \leq i \leq |\texttt{chs}(\psi)|$ let $\psi_i = \mathbb{F}(\delta^m_{\omega(i),T}, Q, \psi_{i-1})$. Then, $\delta^f_T(Q,\psi) = \{(Q,\psi') \mid \psi' \in \psi_{|\texttt{chs}(\psi)|}\}$*

Like coinductive failure, coinductive success must also be modified to work with predicate programs. In this case, s(ASP) must differentiate between CHS entries and ancestors in the call stack which are exact matches for a goal and those which simply unify with it. Two terms are an **exact match** if they can be constructively unified without binding or altering the prohibited value lists of any variables present in either argument.

Definition 120. *Let T_1, T_2 be operational terms. Let Q be a body of goals, and ψ be an operational state. The operational transformation for the exact match, $\delta^e_{T_1,T_2}$ is defined as follows:*

If both T_1 *and* T_2 *are equal,* $\delta^e_{T_1,T_2}(Q,\psi) = \{(Q,\psi)\}$.

Suppose, T_1 *and* T_2 *are both* structures*, but not equal. If* $\texttt{functor}(T_1) = \texttt{functor}(T_2)$ *and* $\texttt{arity}(T_1) = \texttt{arity}(T_2)$*, with* $\psi_0 = \{\psi\}$*, and for every* $1 \leq i \leq \texttt{arity}(T_1)$*,* $\psi_i = \mathbb{F}(\delta^e_{\texttt{arg}_i(T_1),\texttt{arg}_i(T_2)}, Q, \psi_{i-1})$ *where* $\psi_{\texttt{arity}(T_1)} \neq \emptyset$*,* $\delta^e_{T_1,T_2}(Q,\psi) = \{(Q,\psi)\}$.

Suppose that either T_1 *or* T_2 *is a variable, and not equal to each other. Without loss of generality, assume it is* T_1*, and let* $V_1 = \texttt{vars}(\psi)[\texttt{id}(T_1)]$*. If* $\texttt{type}(V_1)$ *is* bound*, then* $\delta^e_{T_1,T_2}(Q,\psi) = \delta^e_{T_3,T_2}(Q,\psi)$*, where* $T_3 = \texttt{binding}(V_1)$*. Otherwise, if* T_2 *is a variable with* $V_2 = \texttt{vars}(\psi)[\texttt{id}(T_2)]$*,* $\texttt{type}(V_2) = \texttt{unbound}$*, and* $\texttt{prohibited}(V_1) = \texttt{prohibited}(V_2)$ *then* $\delta^e_{T_1,T_{2r}}(Q,\psi) = \{(Q,\psi)\}$.

In all other cases $\delta^u_{T_1,T_2}(Q,\psi) = \emptyset$.

For example, given variables X and Y, if both are constrained against 2, then they are an exact match. However, if X is constrained against 2 and Y against 2 and 3 they are unifiable, but not an exact match. The same applies to compound terms: f(X) and f(Y) are an exact match if X and Y are an exact match.

When testing for coinductive success, exact matches will allow success or failure to be deterministic, while other matches will be non-deterministic. Normally, an exact match is enough to determine a cycle, but this is not the case with positive cycles. If a positive cycle is detected via an exact match, a second check is made fo equality. That is, the two terms are equal. So if (f(X)) is the ancestor, f(Y) can be an exact match, but unless X and Y were previously unified, they are not equal. The testing process for a goal `C` is as follows:

1. `C` is tested against the CHS for exact matches. If one is found, deterministically succeed.
2. `C` is tested against each entry `D` in the call stack. The number of negations between `C` and `D` are counted, excluding `C` and `D` themselves.
   (a) If `C` and `D` are an exact match with no intervening negations, if `C` and `D` are equal, fail deterministically. Otherwise proceed as if `C` and `D` are not an exact match.

(b) If `C` and `D` are an exact match with an even, non-zero number of intervening negations, success is deterministic. *An occurs check is needed to identify loop variables. Loop variables are necessary for more completeness.*

3. If no matches are found or all deterministic matches have been exhausted, execute `C` normally.

Definition 121. *Let G be a body of goals, g be a goal, and ψ be some operational state. The operational transformation δ_g^c and two argument function ρ are defined as follows:*

Let f be a function that maps an integer to a member of `callstack`(ψ) *such that if for $i, j \in \mathbb{Z}$, $i < j$ then $f(i)$ comes before $f(j)$ in the sequence. For all $1 \leq i \leq n$, where n is the number of members in* `callstack`(ψ)*, let $\gamma_i = \delta^e_{g,f(i)}(G, \psi)$. For all $1 \leq i \leq n$, where n is the number of members in* `callstack`(ψ)*, let $\gamma'_i = \delta^e_{notg,f(i)}(G, \psi)$.*

- *If there exists $i \in \mathbb{Z}$ such that $\gamma'_i \neq \emptyset$, $\delta_g^c(G, \psi) = \emptyset$ and $\rho(g, \psi) = \bot$.*
- *If there exists $i \in \mathbb{Z}$ such that $\gamma_i \neq \emptyset$, for all $1 \leq j \leq i$,g is a positive literal if and only if $f(j)$ is positive, and g is equal to $f(i)$, then if g is positive $\delta_g^c(G, \psi) = \emptyset$ and $\rho(g, \psi) = \bot$ and $\delta_g^c(G, \psi) = \{(G, \psi)\}$ and $\rho(g, \psi) = \top$ otherwise.*
- *If there exists $i \in \mathbb{Z}$ such that $\gamma_i \neq \emptyset$ and there exists some $1 \leq j \leq i$ where g is negative if and only if $f(j)$ is positive, then $\delta_g^c(G, \psi) = \{(G, \psi)\}$ and $\rho(g, \psi) = f(i)$.*
- *in all other cases, $\delta_g^c(G, \psi) = \{(G, \psi)\}$ and $\rho(g, \psi) = \bot$.*

The use of exact matches in steps 1 and 2(a) is necessary for completeness. With constructive unification, a call would always succeed if it unified with an entry in the CHS and fail if it unified with an entry in the call stack with no intervening negations. However, this behavior could result in solutions being skipped. For example, in step 2(a), exact matches are needed to avoid false positives when detecting positive loops. Consider the rules

```
r(3.14).
r(V) :- r(V2).
```

Were a program containing these rules to be grounded, positive loops would only be present for those cases where V = V2. However, since V2 will always be unbound, r(V) and r(V2) will always constructively unify. As a result, a positive loop would always be detected if ordinary coinductive success were used, leading to failure in all cases. For instance, consider the query r(1). V2 would unify with 1 cause a positive cycle on r(1). In fact, at this point V1 and 1 would be equal. However, if exact matches are used, a potential positive loop will be detected only when V is unbound, and V and V1 will not be equal.Consider the query r(1) in more detail:

1. r(1) will be checked for coinductive failure/success.
2. Since the conditions for immediate success or failure are unmet, r(1) will be added to the call stack and expanded using the second rule, since the first rule will not unify with it.
3. r(V2) will be checked for coinductive failure/success. r(V2) does not exact match r(1) so no possible cycle detected.
4. Since the conditions immediate success or failure are unmet,
5. r(V2) will unify with r(3.14) and succeed.
6. r(1) will succeed, returning the partial stable model {r(1), r(3.14)}.

6.2.2 Even Loops

Even loops (Definition 23) have special significance in the stable model semantics. They indicate that a goal may be either true or false. The s(ASP) algorithm must account for this. As an example, consider the program

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

and the query p(X). Normally, when the query succeeds, p(X1) will be added to the chs, indicating that p(X) is true for all values of X. However, that is only one model out of an infinite number. By the definition of extended stable models there is a model for every subset of the s(ASP) universe.

Now, consider the query p(X),**not** p(a). This query should succeed because X here is existential, but the system does not know that what is really wanted is a proof for X=a. The solution to this are loop variables.

Definition 122. ***Loop variables*** *are variables which occur in both a recursive call and its ancestor in an even loop, and are unbound or negatively constrained when the ancestor call succeeds. They are represented by variable states with type* `loop`.

For a variable to be present in both a recursive call and its ancestor, it must also occur in each of the intervening calls which are part of the even loop. The chain of literals in the loop may all be either true or false for every grounding of the loop variables which does not produce a contradiction in the CHS. Because the s(ASP) universe is infinite, the result is that any program with at least one extended stable model containing a loop variable will have an infinite number of elements, and thus have an infinite number of partial extended stable models. Consider the program

```
p(X, Y) :- not q(X, Y), t(Y, Y).
q(X, Y) :- not p(X, Y).
```

with the query q(X, Y). Because X occurs as a loop variable, q(X, Y) and **not** p(X, Y) may be both true or both false for each possible grounding of X and Y. For example, {q(1, 2), **not** p(1, 2)} and {p(a, b), q(1, 2), **not** p(1, 2), **not** q(a, b)} are both partial extended stable models of the above program. Furthermore, since the domains of X and Y are infinite, there must be an infinite number of these partial stable models.

6.2.3 Consistency Check

Recall from Section 2.7 that any constraints imposed by OLON rules in a program are enforced by appending a special goal, the NMR check, to each query. The NMR check calls sub-checks for each OLON rule in the program, enforcing the constraints which they impose. Two changes are necessary to support predicate programs. First, the NMR sub-checks must be generated using our new dual rule algorithm. Second, because OLON rules apply global constraints, we must ensure that the sub-checks are satisfied for all possible values of their variables.

The first step is to use the new dual rule algorithm presented in Section 6.1.3 when generating NMR sub-checks. A sub-check is created by first appending the negation of an OLON rule's head to its body (if not already present), taking the dual of the modified rule and renaming the head to a unique atom. The program

```
p(X) :- q(X), not p(X).
```

will produce the sub-check

```
chk_p(X) :- not q(X).
chk_p(X) :- q(X), p(X).
```

Since the s(ASP) algorithm is simulating a ground program, there is an instance of the constraint enforced by the OLON rule for each value the variables in the head can be. This is accomplished using the for-all mechanism, also described in Section 6.1.3. The body of the NMR check is modified by considering each variable in a sub-check goal to be universally quantified, and abstracting it with a forall. Thus, the NMR check for the above sub-check would be

```
nmr_check :- forall(X, chk_p(X)).
```

With these modifications, the NMR check will correctly apply any constraints imposed upon the program. However, this does make it more difficult to identify whether a program

is legal without running it. Because each sub-check is a dual rule, the same caveats which apply to calling negated goals, discussed in Section 6.1.3, also apply to the nmr rules.

Unlike the above situations, the initial detection of OLON rules remains unchanged. As explained in Section 2.7, OLON rules are detected by finding cycles in the call graph which contain an odd number of negations. With the addition of variables it would appear to be more complicated. Consider the following rule:

```
p(X) :- q(X, Y), not p(Y).
```

An odd loop is present, but only when X = Y. It seems that such a case would need to account for this. That is, adding a constraint to the resulting sub-check to exclude cases where X = Y. Enforcing this however, would be very computationally intensive, if possible at all. Perhaps needing a full data flow analysis of the program. Fortunately, this is not necessary. Since a check will always succeed if the rule is not OLON. THerfore, we do not even need to consider the variables at all and select candidate OLON rules from the call graph at the predicate level. So, assuming the above is the only potential OLON rule int he program, the nmr check would look like this:

```
nmr_check :- forall(X, chk_p(X)).

chk_p(X) :- forall(Y, chk_p2(X,Y)).

chk_p2(X,Y) :- not q(X,Y).
chk_p2(X,Y) :- q(X,Y), p(Y).
chk_p2(X,Y) :- q(X,Y), not p(Y), p(X).
```

When X = Y, the sub-check will always succeed. One of the first two rules for chk_p2(X, Y) will succeed in cases where the original rule for p(X) would fail and the third clause will succeed in cases where the original rule would succeed. This will always be the case for such "conditional" OLON rules. In cases where no OLON is present, the corresponding sub-check will always be satisfied.

6.3 The Complete Method

The previous sections discussed and presented the various components that make up the s(ASP) algorithm. This section will connect the components together to form the full algorithm.

Informally, a query Q is executed for some program P as follows:

- The Program is Preprocessed to generate a new program P'
  - The call graph of P is examined, OLON rules identified, and nmr check and subcheck rules are generated and combined with P.
  - The nmr goal is appended to the query.
  - The dual rule generation algorithm is used to generate the duals for all predicates.
  - The duals are combined with the result above, generateing P'.
- Each goal, G, in the query is executed in order.
  - If the goal is an arithmetic expression, unification, disunification or forall it is executed as defined.
  - Otherwise, the goal is checked against the CHS.
    - If the CHS contains an exact match for `not G`, `G` fails *deterministically.*
    - If the CHS contains an exact match for `G`, `G` succeeds *deterministically.*
    - If no exact match is present in the CHS, `G` is constrained against any CHS entries which unify with `not G`. This process may be *non-deterministic.*
  - If G passes the CHS check without succeeding or failing, the call stack is checked for cycles. Starting with the most recent element and going backwards:
    - If `G` is an equal to some entry in the call stack with no intervening negations, `G` fails *deterministically* (positive loop).

    * If `G` is an exact match for an entry in the call stack with an even, non-zero number of negations, `G` succeeds *deterministically* (coinductive success).
    * If `G` *constructively unifies* with an entry in the call stack with an even, non-zero number of negations, `G` succeeds *non-deterministically* (coinductive success).
  - If G passes the call stack check without succeeding or failing G is expanded using rules in P'. If the body of the rule that replaces G succeeds then a clean copy of G is added to the CHS.

- If every goal in the query succeeds, then the query succeeds.

Figure 6.1 contains an abstract meta-interpreter for this method, but formally this algorithm will be defined using operational transformations.

At the top level, we need two transformations. The first to represent a "step" in the algorithm. That is, replace a goal with the body of a matching rule, and one that repeats the process until the goal list is empty. In these definitions it will be assumed that operators such as "is", "==", and "\=" are syntactic sugar for predicates, and therefore we will only look at goals that can be unified with a head and forall. It will also be assumed that the initial program is preprocessed and contains dual rules and NMR checks.

Definition 123. *For some program P, body of goals G, and operational state ψ, $\delta^{\downarrow}$ is defined as follows:*

If G is empty then $\delta^{\downarrow}(G, \psi) = \emptyset$. Otherwise, let g be the minimal goal of G, $G' = G \setminus \{g\}$, and $P' = \theta_P(g)$.

if $\texttt{functor}(g) = forall$ **and** $\texttt{arity}(g) = 2$**:** *Let $\gamma = \delta^{\downarrow}(\{\texttt{arg}_2(g)\}, \psi)$. Let ξ be a one-to-one function that maps a prohibited value list, T, and a natural number between 1 and $|T|$ to some element of T. The actual definition of ξ does not matter. Let ϕ be a function*

such that given a state, ψ_1, prohibited value list and a natural number,i, $\phi(\psi_1, T, i) = \{\psi_1\}$ when $i = 0$, and when $i \neq 0$ $\phi(\psi_1, T, i) = \{\psi_2 \mid \psi_3 \in \phi(\psi_1, T, i-1), (\emptyset, \psi_2) \in \delta^{\downarrow}(\{\mathtt{arg}_2(g)\}, \mathtt{arg}_1(g) \rightarrow_{\psi_3} \xi(T, i))\}$. For convenience, let $\phi(\psi, T) = \phi(\psi, T, |T|)$. Then

$$\begin{aligned}\delta^{\downarrow}(G, \psi) = \{(G', \psi') \mid\ & (\emptyset, \psi_1) \in \gamma, \\ & \psi_2 \in \phi(\psi_1, \mathtt{prohibited}(\mathtt{vars}(\psi_1)[\mathtt{id}(\mathtt{arg}_1(g))])), \\ & \psi' = \psi_2\langle \mathtt{arg}_1(g)[\mathtt{prohibited} = \emptyset, \mathtt{type} = \mathtt{bound}], \mathtt{clean}(\mathtt{arg}_2(g)) \rightarrow \mathtt{chs}\rangle\}\end{aligned}$$

Otherwise, *Let $\gamma = \{X \mid T \in \mathtt{chs}(\psi), X \in \delta^e_{T,g}(G', \psi)\}$. If $\gamma \neq \emptyset$ an exact match was found so $\delta^{\downarrow}(G, \psi) = \{(G', \psi)\}$. On the other hand, if $\gamma = \emptyset$ no exact match was found. Let $\gamma' = \{X \mid T \in \mathtt{chs}(\psi), X \in \delta^e_{T,not\ g}(G', \psi)\}$. If $\gamma' \neq \emptyset$ then an exact match of the negation was found. So, $\delta^{\downarrow}(G, \psi) = \{\}$. Let $\lambda = \delta^f_g(G', \psi)$, and $\lambda' = \{(\rho(g, \psi_1), \psi_3) \mid (G', \psi_1) \in \lambda, (G', \psi_2) \in \delta^c_g(G', \psi_1)\}$ Now, define set Δ as follows: For all $(R, \psi') \in \lambda'$,*

- *if $R = \top$, $(G', \psi'\langle \mathtt{clean}(g) \rightarrow \mathtt{chs}\rangle) \in \Delta$),*
- *if $R \neq \top$ and $R \neq \bot$, with V being the set of variables that are in both g and R, $\{(G', \psi'\langle \mathtt{cloop}(V, g) \rightarrow \mathtt{chs}\rangle\} \subseteq \Delta$, and*
- *otherwise, $(G', \psi'') \in \Delta$ if $\psi'' \in \{\psi'' \mid r \in \theta_P(g), (\emptyset, \psi_1) \in \delta^u_{g,\mathtt{head}(r)}(\emptyset, \psi'), (\emptyset, \psi'') \in \delta^{\downarrow}(\mathtt{body}(r), \psi_1)\}$*

At the foundation, the algorithm is about proving a body of goals. The query is a body of goals, and when expanding a goal, the goal is replaced with a body of goals. Since this behavior is need in multiple places, we will define a function to handle it. This function will map an operational transformation, body of goals, and an operational state to a set of pairs of body of goals and operational states.

Definition 124. *Let P be a program. Let δ be an operational transformation, Q be a body of goals, and ψ be an operational state. The function $\mathbb{D}$ is defined as follows:*

If $Q = \emptyset$, then $\mathbb{D}(\delta, Q, \psi) = \{(Q, \psi)\}$. Otherwise, $\mathbb{D}(\delta, Q, \psi) = \{X \mid (Q', \psi') \in \delta(Q, \psi), X \in \mathbb{D}(\delta, Q', \psi')\}$.

Definition 125. *For some program P, body of goals G, and operational state ψ, $\delta^{\downarrow}_{P}$ is defined as follows:j*

If G is empty then $\delta^{\downarrow}_{P}(G,\psi) = \{(G,\psi)\}$. Otherwise, let g be the minimal goal of G, $G' = G \setminus \{g\}$, and $P' = \theta_P(g)$.

if $\texttt{functor}(g) = forall$ **and** $\texttt{arity}(g) = 2$**:** *Let $\gamma = \delta^{\downarrow}_{P}(\{\texttt{arg}_2(g)\},\psi)$. Let ξ be a one-to-one function that maps a prohibited value list, T, and a natural number between 1 and $|T|$ to some element of T. The actual definition of ξ does not matter. Let ϕ be a function such that given a state, ψ_1, prohibited value list and a natural number,i, $\phi(\psi_1,T,i) = \{\psi_1\}$ when $i = 0$, and when $i \neq 0$ $\phi(\psi_1,T,i) = \{\psi_2 \mid \psi_3 \in \phi(\psi_1,T,i-1), (\emptyset,\psi_2) \in \delta^{\downarrow}_{P}(\{\texttt{arg}_2(g)\}, \texttt{arg}_1(g) \rightarrow_{\psi_3} \xi(T,i))\}$. For convenience, let $\phi(\psi,T) = \phi(\psi,T,|T|)$. Then*

$$\begin{aligned}\delta^{\downarrow}_{P}(G,\psi) = \{(G',\psi') \mid\ & (\emptyset,\psi_1) \in \gamma,\\ & \psi_2 \in \phi(\psi_1, \texttt{prohibited}(\texttt{vars}(\psi_1)[\texttt{id}(\texttt{arg}_1(g))])),\\ & \psi' = \psi_2\langle \texttt{arg}_1(g)[\texttt{prohibited} = \emptyset, \texttt{type} = \texttt{bound}], \texttt{clean}(\texttt{arg}_2(g)) \rightarrow \texttt{chs}\rangle\}\end{aligned}$$

Otherwise, *Let $\gamma = \{X \mid T \in \texttt{chs}(\psi), X \in \delta^{e}_{T,g}(G',\psi)\}$. If $\gamma \neq \emptyset$ an exact match was found so $\delta^{\downarrow}_{P}(G,\psi) = \{(G',\psi)\}$. On the other hand, if $\gamma = \emptyset$ no exact match was found. Let $\gamma' = \{X \mid T \in \texttt{chs}(\psi), X \in \delta^{e}_{T,not\ g}(G',\psi)\}$. If $\gamma' \neq \emptyset$ then an exact match of the negation was found. So, $\delta^{\downarrow}_{P}(G,\psi) = \{\}$. Let $\lambda = \delta^{f}_{g}(G',\psi)$, and $\lambda' = \{(\rho(g,\psi_1),\psi_3) \mid (G',\psi_1) \in \lambda, (G',\psi_2) \in \delta^{c}_{g}(G',\psi_1)\}$ Now, define set Δ as follows: For all $(R,\psi') \in \lambda'$,*

- *if $R = \top$, with $(g',\psi'') = \texttt{clean}(g,\psi')$, $(G',\psi''\langle g' \rightarrow \texttt{chs}\rangle) \in \Delta)$,*
- *if $R \neq \top$ and $R \neq \bot$, with V being the set of variables that are in both g and R, with $(g',\psi'') = \texttt{cloop}(V,g,\psi')$, $\{(G',\psi''\langle g' \rightarrow \texttt{chs}\rangle\} \subseteq \Delta$, and*
- *otherwise, $(G',\psi'') \in \Delta$ if $\psi'' \in \{\psi'' \mid r \in \theta_P(g), (\emptyset,\psi_1) \in \delta^{u}_{g,\texttt{head}(r)}(\emptyset,\psi'), (\emptyset,\psi'') \in \mathbb{D}(\delta^{\downarrow}_{P}, \texttt{body}(r), \psi_1)\}$,*

and nothing else is in Delta. Then $\delta_P^{\downarrow}(G, \psi) = \Delta$.

Finally, putting them together we can compute the partial models of a program.

Definition 126. *Let* P *be a program, with* P' *being the preprocessed program with dual rules and nmr checks. Let nmr be the zero-arity predicate representing the nmr check. Let* Q *be a body of goals with an instance of nmr as its maximal goal. Let* ψ *be the operational state such that* $\texttt{callstack}(\psi) = \emptyset$, $\texttt{chs}(\psi) = \emptyset$, *and operational variable* V *is in* Q *if and only if* V *is in* $\texttt{vars}(\psi)$.

Then, $\{\texttt{chs}(\psi') \mid (G, \psi') \in \mathbb{D}(\delta_{P'}^{\downarrow}, Q, \psi)\}$ *is the set of all partial extended stable models of* P.

```
sasp(Q, NMR) :-
  append(Q, NMR, Q2),
  exec_goals(Q2),
  check_loop_variable_domains,
  print_chs.

exec_goals([X | T]) :-
  exec_goal(X),
  exec_goals(T).
exec_goals([]).

exec_goal(X) :-
  X = forall(V, G), !,
  exec_forall(V, G).
exec_goal(X) :-
  check_chs_and_call_stack(X, CHSResult),
  exec_goal2(X, CHSResult).

exec_goal2(X, success). % coinductive success
exec_goal2(X, expand) :- % expand using rules
  get_matching_rule(X, R),% select subsequent
                          % rules on backtracking
  exec_goals(body(R)),
  add_to_chs(X).
```

Figure 6.1: Abstracted s(ASP) Meta-interpreter.

```
exec_forall(V, G) :-
  unbound(V), % fail if variable is bound
                % or constrained
  exec_goal(G), % first solve goal normally
  Cons = get_constraints(V), % fail if variable
                             % is bound
  exec_with_each_constraint_value(G, V, Cons),
  set_unbound(V), % goal succeeded for all V.
  add_to_chs(G).

check_chs_and_call_stack(X, failure) :-
  Xn = dual(X),
  exact_match_in_chs(Xn), !. % coinductive failure
                             % unavoidable
check_chs_and_call_stack(X, success) :-
  exact_match_in_chs(X), !. % coinductive success
                            % unavoidable
check_chs_and_call_stack(X, Result) :- % avoid failure,
                                       % if possible.
  constrain_goal_against_unifiable_duals(X), % non-deterministic.
  check_call_stack(X, Result).

% Check the call stack for cycles on current goal.
% If a cycle over an exact match is found, don't
% look for other matches when backtracking.
check_call_stack(X, failure) :-
  call_stack_has_positive_cycle_w_exact_match(X), !.
check_call_stack(X, success) :-
  call_stack_has_even_cycle_w_exact_match(X), !.
check_call_stack(X, success) :-
  call_stack_has_even_cycle(X), % non-deterministic
  unify_goal_w_match.
check_call_stack(X, expand).
```

Figure 6.1: Abstracted s(ASP) Meta-interpreter Continued.

6.4 Understanding the Partial Model

The result of the s(ASP) algorithm is a set of partial models. It is our claim, which will be proven in Chapter 7, that each partial model represents some subset of an extended stable

model for the program. As can be seen from definition 126, a partial model is the CHS after the query has been executed. The CHS is a set of goals, and is not necessarily ground.

Since extended stable models are all propositional, the partial model must also represent some set of propositions. As stated in Chapter 2, A propositional program can be generated from a program by grounding the program and treating each ground term as a proposition. The same can be said for partial models.

Each `unbound` variable in the CHS is universally quantified. While `loop` variables are a special case of existentially quantified variables. They can be considered universally quantified over some subset of the variable's domain. All subsets that do not lead to an empty variable or contradiction in the CHS may be used. Since the domains of all variables are infinite, there will be an infinite number of valid subsets to chose from. Therefore, such partial models represent an infinite number of grounded partial models.

CHAPTER 7

PROOFS AND APPLICATIONS FOR S(ASP)

This chapter will prove the correctness of the s(ASP) algorithm, discuss completeness, and present several applications of s(ASP) that have been produced. As with the last chapter, we will assume that all program executions do not violate the restrictions, and the algorithm's behavior is undefined in the case the restrictions are violated.

The work presented in this chapter was equally shared with Dr. Kyle Marple. All theory and design was a group effort. All definitions and proofs are the work of this author. While much of the descriptive text was based off of Dr. Marple's work. The s(ASP) implementation was also the work of Dr. Marple.

7.1 Completeness

While s(ASP) is sound for programs and executions that match our assumptions, completeness for this set is impossible. While we have strived to make s(ASP) complete for the largest class of programs possible, we leave the precise definition of this class and the associated proofs to future work. Instead, we argue that the utility of our method outweighs its lack of completeness.

It's easy enough to show that s(ASP) cannot possibly be complete for all legal programs. Consider the class of *stratified programs*. That is programs with no loops over negation. A stratified program will always have a unique stable model which coincides with its *perfect model*, the model produced by the perfect model semantics (Cadoli and Schaerf, 1993). However, the perfect model of such a program may be incomputable (Apt and Blair, 1990). Therefore, even though a stable model must exist for such a program, we may be unable to compute it. As such, it is not possible to guarantee completeness for all programs.

The trade-off for this loss of completeness is a massive increase in expressive power. The propositional stable model semantics can only express relations which are co-NP, however,

the predicate stable model semantics and s(ASP) can express relations which are Π_1^1 (Schlipf, 1990; Cadoli and Schaerf, 1993).

In addition to increased computational expressiveness, s(ASP) supports lists, complex data structures and real numbers, providing programmers with tools not found in any other implementation of the stable model semantics. With these features, even programs which can already be expressed in the propositional semantics may be easier to write in s(ASP).

Aside from completeness itself, the only "desirable" property that we lose compared to other implementations of the stable model semantics is the guarantee that a program will always terminate. However, guaranteed termination is a double-edged sword. It implies that only decidable problems can be encoded, as, by definition, this guarantee cannot be applied to semi-decidable or undecidable problems. By abandoning guaranteed termination, we are able to support programs which encode such problems, something that no other implementation of the stable model semantics can claim. This is significant, as problems which are undecidable in general may still produce useful results for some cases.

Thus, we trade completeness for superior functionality and the ability to encode problems which no other implementation of the stable model semantics can handle. While completeness is certainly desirable, it is our firm belief that the gains derived from this trade significantly outweigh the losses.

7.2 Proof of Soundness

Informally, the s(ASP) algorithm is sound if every partial model it generates is part of some stable model of the program. To show this we must first separate the (grounded) program into three parts:

- The set of rules required to prove the partial model.
- The set of rules related to the query but not needed to prove the partial model.
- The set of rules not related to the query.

We will first show that the second set of rules can be removed without affecting the result. That is, the partial model is a part of some stable model of the new program, and that stable model is a stable model of the original program. We also show that rules needed to prove that some literal is false can be modified by removing goals as long as one "false" goal remains. Using these two modifications and the splitting theorem (Lifschitz and Turner, 1994), we can isolate a subprogram comprised only of rules and literals touched by the s(ASP) algorithm.

Using the terminology of (Lifschitz and Turner, 1994), detailed in the next section, this subprogram can be considered the "bottom" of the program, with the remaining rules forming the "top" of the program. We show that the partial model generated by the s(ASP) algorithm is a stable model of the bottom of the program. Then, in accordance with the splitting theorem we transform the top of the program and show that since the NMR check is satisfied a stable model exists for it. By using the splitting theorem and the points discussed above, we show that the union of the partial model and the top's stable model is a stable model the original program.

Before continuing, we define the following for convenience:

Definition 127. *Let* `G` *be a goal constructed from atom* `A`*. Then,* `atom(G)` $=$ `A`*, and* `goal(A)` *is the set of both goals (positive and negative) that can be constructed from* `A`*. The arguments of* `A` *are said to be the arguments of* `G`*.*

Definition 128. *Let* `R` *be a rule of the form:*

p `:-` q_1`, ...,` q^i`, ...,` q_m`,`

`not` r_1`, ...,` `not` r_j`, ...,` `not` r_n`.`

Then:

- ***head(R)*** $= p$
- ***pos(R)*** $= \{\ q_1, \ldots, q_i, \ldots, q_m\ \}$
- ***neg(R)*** $= \{\ not\ r_1, \ldots, not\ r_j, \ldots, not\ r_n\ \}$
- ***lit(R)*** $= \{\ H\ \} \cup pos(R) \cup neg(R)$

7.2.1 Review of Splitting Theorem

Next, we review the splitting theorem (Lifschitz and Turner, 1994). A set of literals can be used to *split* a ground program. This set is called a *splitting set*, and is defined as follows.

Definition 129. *A set of ground atoms U is a* ***splitting set*** *for some ground program P if for every rule R in P, $head(R) \in U \Rightarrow lit(R) \subseteq U$.*

The program is divided into two parts, the top and the bottom. The bottom is the set of rules related to the splitting set and the top is the set of all other rules.

Definition 130. *Let P be a ground program and U a splitting set for P. The* ***bottom*** *of P with respect to U, specified as $b_U(P)$, is the set of rules r for which $lit(r) \subseteq U$. The set $P \setminus b_U(P)$ is the* ***top*** *of P with respect to U.*

The stable models of a ground program $\mathtt{P}$ can be computed by combining the stable models of $b_U(\mathtt{P})$ and the stable models generated by the top. This requires us to generate a new program from $\mathtt{P} \setminus b_U(\mathtt{P})$ based on the stable models for $b_U(\mathtt{P})$.

Theorem 25. *Let P be some ground program, U a splitting set of P, and X a stable model for $b_U(P)$. For each rule r in P such that $pos(r) \subset X$ and $neg(r) \cap X = \emptyset$, we define a new rule r' with:*

- *$head(r') = head(r)$,*

- $pos(r') = pos(r) \setminus U$
- $neg(r') = neg(r) \setminus U$

We define the program $e_U(P \setminus b_U(P),\ X)$ *as the set of of all such new rules, and for some stable model* Y *of* $e_U(P \setminus b_U(P), X)$, $X \cup Y$ *is a stable model of* P.

If either $b_U(P)$ *or* $e_U(P \setminus b_U(P), X)$ *has no stable model then there is no stable model for* P.

Proof. Proofs for these results are available in the original paper which introduced the splitting theorem (Lifschitz and Turner, 1994). □

7.2.2 Stripping Unneeded Rules and Body Literals

Lemma 11. *Let* P *be a ground program, and* M *an stable model of* P. *Let* R *be a rule not in* P *such that the head of* R *is in* M. *Then,* M *is a stable model of the program* $P \cup \{\ R\ \}$.

Proof. There are two cases for R:

Case 1: There exists $\mathtt{L} \in \text{neg}(\mathtt{R})$ such that $\mathtt{L} \in \mathtt{M}$. In this case R is removed when computing the reduct, and the reduct does not change. Therefore, M is the least model of the reduct.

Case 2: R is not removed when the reduct is computed. We will call the transformed rule $\mathtt{R}'$.

Since the head of R is in M there must exist some rule R_2 in P with $\text{head}(\mathtt{R}) = \text{head}(\mathrm{R}_2)$ such that for all $\mathtt{L} \in \text{pos}(\mathrm{R}_2)$, $\mathtt{L} \in \mathtt{M}$ and for all $\mathtt{L} \in \text{neg}(\mathrm{R}_2)$, $\mathtt{L} \notin \mathtt{M}$. We will call the transformed rule in the reduct $\mathtt{R}''$.

The only way $\mathtt{R}'$ can affect the least model of $\mathtt{P} \cup \{\ \mathrm{R}\ \}$ is to be used to place its head in it. Therefore, since $\mathtt{R}'$ can not affect literals besides its head, the body literals in $\mathtt{R}''$

must be in the least model of the reduct of $\mathtt{P} \cup \{ \mathrm{R} \}$. Thus, head($\mathtt{R}''$) (which is also head(R′)) must also be in the least model, and the least model for the reduct of $\mathtt{P} \cup \{ \mathrm{R} \}$ is the same as for P's reduct. So, M must be the least model of the reduct of $\mathtt{P} \cup \{ \mathrm{R} \}$.

Thus, M is a stable model of $\mathtt{P} \cup \{ \mathrm{R} \}$.

□

Lemma 12. *Let P be a ground program and M a stable model of P. Let R be a rule in P such that there exists a literal in the body that is not in M, and let G be a ground goal. Let P′ be the program constructed by adding G to the body of R. M is a stable model of P′.*

Proof. Let P be a ground program, and M be a stable model of P. Let R be a rule in P such that the head of R is not in M. Let G be some ground goal.

Create new program P′ by adding G to the body of R. Call this rule R′.

We have two cases:

1. There exists some literal $\mathtt{L} \in \mathrm{neg}(\mathtt{R})$ and $\mathtt{L} \in \mathtt{M}$, or

2. there is some literal $\mathtt{L} \in \mathrm{pos}(\mathtt{R})$ such that $\mathtt{L} \notin \mathtt{M}$.

In case 1, we know that R′ will be removed when computing the reduct, and thus can not affect the least model. So we only need to consider case 2. In addition we can assume that G does not cause the rule to be removed from the reduct(otherwise it can not affect the least model). Now, notice that the addition of G does not affect the truth value of L. Thus, it is possible that the same process that causes L to not be in the least model of the reduct of P will cause it to not be in the reduct of P′. So, R′ cannot be used to place its head in the least model. Since no other rules have changed, the least model of the reduct of P′ is the same as the one for P, and thus M is a stable model of P′. □

To prove theorem 26, we want to separate the part needed to prove the query from the rest. Since the s(ASP) algorithm is goal directed we need to remove rules related to the query that are not needed to prove it and goals in rules related to the query, but not needed to prove it. We call this process **trimming**. Then we will make use the splitting theorem from (Lifschitz and Turner, 1994) to divide the new program into two parts, treating the portion needed to prove the query as the bottom, and the rest as the top.

It is important to remember that the s(ASP) algorithm works directly with the ungrounded program, but we will be trimming its ground program. When executing a rule (using it to prove some goal) it is possible (and likely) that variables in the rule will be constrained or bound by some goals in its body. These changes can be tracked via the operational states.

An operational transformation operates on a body of goals and an operational state, producing a set of such pairs. These pairs are then operated on to produce new sets that are unioned together. The operation transformation $\delta_P^{\downarrow}$ can be used to define a tree of these state changes.

Definition 131. *Let P be a program, Q be a body of goals, and M be a partial model generated by the s(ASP) algorithm for Q. A tree can be generated by tracing backwards through $\delta_P^{\downarrow}$.*

- *Each element of M is a leaf node and associated with either a fact or marked as a coinductive success, and the current operational state.*
- *Each node has a parent, if it was the minimal goal of a body of goals generated via rule in $\delta_P^{\downarrow}$. It is associated with that rule and operational state after the rule body was proved.*

Finally, all root nodes of the above trees are children of a new root node associated with Q.

Let this tree be calls $\mathbb{T}_{Q,M}$.

Definition 132. *To* ***trim*** *a program we will follow the following algorithm. Let* `P` *be a program,* `P'` *be the result of grounding* `P` *over the s(ASP) universe,* Q *a body of goals,* `M` *a partial model of* `P` *generated by the s(ASP) algorithm for* Q*, and* `M'` *the grounding of* `M` *over the s(ASP) universe.*

1. *Using* $\mathbb{T}_{Q,M}$*, for all nodes whose predicates are not internally generated, mark all rules in* `P'` *is a grounding of the associated rule with respect to* ψ.

2. *After* `M` *is computed: Create a new program* P'' *from* `P'` *by:*

   - *removing all rules* `R` *from* `P'` *for which the head(*`R`*)* $\notin$ `M'` *and* `R` *is not marked, and*
   - *transform all rules* `R` *in* `P'` *for which not head(*`R`*)* $\in$ `M'` *by removing all body goals for which nether they nor their negations are in* `M`.

3. P'' *is the result of* ***trimming*** `P'` *with respect to* `M'`.

7.2.3 Forall

Lemma 13. *Let* `P` *be a s(ASP) program,* `G` *be an atom, and* `X` *an unconstrained variable in* `G`*. Let* $\mathcal{G}$ *be the set of all goals obtained by grounding* `X` *in* `G` *over the s(ASP) universe. Then, a* `forall(X, G)` *in* `P` *succeeds if and only if all goals in* $\mathcal{G}$ *succeed.*

Proof. Assume the opposite is true. That is, either `forall(X, G)` succeeds and some `L` $\in \mathcal{G}$ fails, or all goals in $\mathcal{G}$ succeed, but `forall(X, G)` fails.

Case 1: Suppose `forall(X, G)` succeeds, but there exists some `L` $\in \mathcal{G}$ such that `L` fails. There are two phases for the forall to succeed. First, `G` must succeed with `X` unbound. Then prove `G` with `X` grounded with each of its constraints. Since `L` fails, the value corresponding to `X` in `L` could not have been in the constraint list. Otherwise, the forall

would have failed in the second phase. However, we could take the proof tree generated by the first phase, and ground `X` to obtain a proof tree for `L`, meaning there is a way for `L` to succeed. A contradiction.

Case 2: Suppose `forall(X, G)` fails, but all $\texttt{L} \in \mathcal{G}$ succeed. We know that the forall could not have failed in the second phase, otherwise there would be some $\texttt{L} \in \mathcal{G}$ such that `L` fails. Therefore there are two possibilities. Either, there is no way for `G` to succeed or all ways require `X` to be ground. The second case cannot be the case since all $\texttt{L} \in \mathcal{G}$ succeed, and by definition, the s(ASP) universe contains an infinite number of terms that do not appear in the herbrand universe of `P`. Thus there exists some term in the s(ASP) universe for which `X` cannot be explicitly grounded against. The first case also cannot be true since all $\texttt{L} \in \mathcal{G}$ succeed, thus there must be a way for `G` to succeed. A contradiction.

Therefore, `forall(X, G)` succeeds if and only if all $\texttt{L} \in \mathcal{G}$ succeeds. □

7.2.4 Constructive Coinductive Failure

Lemma 14. *Let `P` be a program, `G'` a goal currently in the CHS, and `G` be a goal that we wish to prove. Let $(Q, \psi) \in \delta^f_{\texttt{c}}$. Then, with respect to ψ, `G` and `G'` do not unify.*

Proof. The coinductive failure operational tranformation merely applies the mismatch transformation to `G` and each element of the CHS. Therefore it is enough to show that the result of the mismatch cannot unify with $\texttt{G}'$.

Let Q be some conjunction of goals. Let ψ, ψ' be some operational states such that $(Q, \psi') \in \delta^m_{\texttt{c}',\texttt{c}}(Q, \psi)$.

Without loss of generality, assume $\texttt{value}_\psi(\texttt{G}) = \texttt{G}$ and $\texttt{value}_\psi(\texttt{G}') = \texttt{G}'$ and

Firstly, $\texttt{G}'$ and `G` cannot equal, otherwise $\delta^m_{\texttt{c}',\texttt{c}}(Q, \psi)$ would be empty. If They have differing functors or arity they cannot unify. So, we only need to consider the case where they have

the same functor and arity, but are not equal. In this case, we must show that there exists some $1 \leq i \leq |\texttt{arity}(\texttt{G})|$ such that $\delta^{u}_{\texttt{arg}_i(\texttt{G}),\texttt{arg}_i(\texttt{G}')}(Q, \psi') = \emptyset$. In order for $\delta^{m}_{\texttt{G}',\texttt{G}}(Q, \psi)$ to not be empty there must exist some $1 \leq i \leq |\texttt{arity}(\texttt{G})|$ such that $\delta^{m}_{\texttt{arg}_i(\texttt{G}'),\texttt{arg}_i(\texttt{G})}(Q, \psi) \neq \emptyset$.

We will show this by inducting over the depth of the term. The depth of a variable or constant is zero, and the depth of a structure with non-zero arity is one more than the maximal depth of all its arguments.

Base Case. Assume $\texttt{arg}_i(\texttt{G}')$ has depth 0. If both are constant then they cannot be equal, and therefore do not unify. If $\texttt{arg}_i(\texttt{G}')$ is an `unbound` variable then either $\texttt{arg}_i(\texttt{G})$ is a constant and in the prohibited value list or a variable that has been bound to something in the prohibited list. In both cases, they cannot unify. Finally, $\texttt{arg}_i(\texttt{G}')$ could be a `loop` variable. In this case the terms would be disunified, and by definition cannot unify.

Inductive hypothesis. Let k be a natural number. Assume $\texttt{G}$ does not unify with $\texttt{G}'$ if $\texttt{G}'$ has a depth less than or equal to k.

Inductive Step. Assume $\texttt{G}'$ has a depth of $k + 1$. Therefore it must be a structure with non-zero arity. If $\texttt{G}$ is a structure with a different functor or arity then they cannot unify. If $\texttt{G}$ is a variable then either $\texttt{G}'$ will be in its prohibited list (meaning they cannot unify) or will be bound to a clean copy of $\texttt{G}'$ and treated as a structure with the same functor and arity. If $\texttt{G}$ is a structure with the same functor and arity, then there must be an argument that mismatched with the corresponding argument of $\texttt{G}'$ and by the inductive hypothesis, they cannot unify. Therefore, $\texttt{G}$ and $\texttt{G}'$ cannot unify.

□

7.2.5 Main Proof

Theorem 26. *Let P be a program, and M a partial model of P generated by the s(ASP) algorithm. Let M_2 and P_2 be the results of grounding M and P, respectively, over the s(ASP) universe. There exists a stable model X of P_2 such that for all literals L in M_2, L is in X, and for all literals L with not L in M_2, L is not in X.*

Proof of Theorem 26. Let P be a program, and M a partial model of P generated by the s(ASP) algorithm. Let M_2 and P_2 be the results of grounding M and P, respectively, over the s(ASP) universe. If M contains loop variables then we may choose a domain for each loop variable that does not contradict the rest of M without loss of generality. This is just selecting one out of the infinite number of partial models represented by M. Assume that there exists at least one assignment that contains no empty variables since we consider such a situation as a failure.

Before proving our claim we must show that M_2 is consistent. That is, for some literal L it is not the case that L and not L are both in M_2. First, notice that if the value of L depends on not L (and visa versa) then the s(ASP) algorithm will fail or the goal will be constrained so that L and not L will not be in the grounding. This is because it is an odd cycle over negation. So we only need to consider the case where $G \in$ goal(L) unifies with something in M, but we want to prove a goal that unifies with the negation of G. However, by lemma 14 we know that the second goal will be restricted so that it no longer unifies with L or not L. So, M_2 is consistent.

Let P_3 be the result of trimming P_2 with respect to M_2, and S be a set of literals such that $L \in S \iff L \in M_2 \wedge$ not $L \in M_2$. S is a splitting set of P_3. Now we must do two things. First we must show that M_2 is a stable model of the bottom, and that there exists a stable model for the top.

To prove that M_2 is a stable model of the bottom we must show that:

1. All literals in $\mathtt{M}_2$ are in the least model of the reduct for $\mathtt{P}_3$, and

2. No literal `L` with `not L` in $\mathtt{M}_2$ will be in it.

Case 1: For all literals $\mathtt{L} \in \mathtt{M}_2$: Let $\mathtt{L}'$ be the non-ground atom in `M` that is used to generate `L`, and `R` be the rule in `P` that is used to prove $\mathtt{L}'$. We can construct a tree by using $\mathtt{L}'$ as the root, and the body literals from `R` as the children. The negated goals will be in `M` and later removed from the rule when computing the reduct. So, we can ignore them and only consider literals as children. Additionally, we will keep the groundings and constraints of the variables at the time of success. Then ground the tree such that the resulting tree has `L` as the root. This corresponds to a rule in $\mathtt{P}_3$, since it would have been marked and therefore not removed. The leaves of such a tree must have facts in the reduct of $\mathtt{P}_3$, and therefore will be in the least model. From there we know that the root of each level going up the tree will be in the least model, including `L`.

Case 2: For all literals `L` such that not $\mathtt{L} \in \mathtt{M}_2$ we must show that there is no way `L` can be in the least model of the reduct for $\mathtt{P}_3$. Firstly, if a rule with `L` as the head is part of a positive cycle for `L`, then it cannot be used to put `L` into the least model. So, we only need to consider noncyclic cases. For these cases we will prove it inductively, and to do that we will define the **level** of a rule. If a rule is a fact then it has level zero. For non-fact rules, we say that a body goal `B` has a level equal to that of the highest level rule with atom(`B`) as the head. The level of all non-facts is one plus the highest level of the body literals. In the case of a body literal for which there are no rules, it is considered level zero.

Base Case: Let `L` be a literal such that `not L` $\in \mathtt{M}_2$. There cannot be a fact for `L`, otherwise `not L` could not be in $\mathtt{M}_2$. The goal not `L` comes from the success of a dual rule, which would always fail if a fact for L existed. If `L` has no rules, then it cannot be in the least model.

Inductive Hypothesis: Let L be a literal such that not $\mathtt{L} \in \mathtt{M}_2$. Suppose all rules with a level less than or equal to k cannot be used to place L in the least model.

Inductive Step: Let L be a literal such that not $\mathtt{L} \in \mathtt{M}_2$. Let R be a rule with L in the head and a level of $\mathtt{k}+1$. In order for the dual to succeed and allow not L to be in the grounding there must be a goal G in R such that $\mathtt{G} \notin \mathtt{M}_2$. Since $\mathtt{G} \notin \mathtt{M}_2$ but was not trimmed from R the negation of G must be in $\mathtt{M}_2$. If G is negated then R would be removed when computing the reduct. So, we only need to consider the case G is not negated. G must have a level of at most k, and by the inductive hypothesis we know that there is no way to place G into the least model of the reduct, and therefore R cannot be used to place L into the least model.

Thus, by induction L is not in the least model of the reduct for $\mathtt{P}_3$.

Therefore, $\mathtt{M}_2$ is a stable model of the bottom of $\mathtt{P}_3$ with respect to the splitting set S.

Now we must show that there exists a stable model for the top. We will do this by observing that the only way for there not to be a stable model is if there is an inconsistency, and there can be an inconsistency only if there is an odd cycle. So, we will show that the modified program from the top will contain no odd cycles.

Let R be an OLON in $\mathtt{P}_2$, and $\mathtt{R}'$ the rule in P such that when grounding $\mathtt{R}'$ over the s(ASP) universe, R is generated. Since R is part of an odd cycle, $\mathtt{R}'$ is also considered part of an odd cycle since we only look at predicate name and arity. Thus there will be a NMR check for $\mathtt{R}'$. By lemma 13, we can treat the NMR check as a conjunction of checks with the head grounded over the s(ASP) universe. So, either there exists a body literal in R with its negation in $\mathtt{M}_2$ or the head of R is in $\mathtt{M}_2$. In the second case, R will either be removed through trimming or will be in the bottom of $\mathtt{P}_3$. For the first case, assume R is not removed through trimming or in the bottom of $\mathtt{P}_3$. Then R will be removed when computing the

partial evaluation for the top since the negation of some literal in the body is in $\mathtt{M}_2$. Thus, there are no odd cycles when computing the answer sets of the top.

It is apparent from the splitting theorem that if $\mathtt{L} \in \mathtt{M}_2$ is a literal then L is in X. So, we only need to show that if not $\mathtt{L} \in \mathtt{M}_2$ then $\mathtt{L} \notin \mathtt{X}$. First, notice that the truth value of L is determined by the bottom of $\mathtt{P}_3$ with respect to S, and cannot be in the stable model of the top. Thus, L cannot be in X.

By lemma 11, we know that a stable model for $\mathtt{P}_3$ is also a stable model of $\mathtt{P}_2$. □

7.3 Implementation and Examples

A fully functional prototype implementation of the method presented here has been created, also using the name s(ASP). The implementation is written in Prolog and totals about 4,300 lines of code (excluding comments and blank lines). An open source release is available at (Marple, 2015).

Unlike its predecessor, Galliwasp, s(ASP) is completely self-contained: neither grounder nor separate compiler is required. As with our method, the prototype will accept any legal normal logic program and execute it *without grounding any portion of the program at any stage*. While the prototype is not designed to be competitive in terms of speed, our method allows it to offer features not found in any other implementation of the stable model semantics, including answer set programming systems. In the following subsections, we will look at how s(ASP) behaves with two number of examples.

7.3.1 Example: N Queens with Lists

A variant of the N queens problem using lists, can be found in Figure 7.1. This example is of particular interest, as it has no finite grounding and thus cannot be run by other implementations of the stable model semantics. Additionally, the even loop in the last two lines of the code will produce two loop variables, discussed in Section 6.2.2.

```
% solve the N queens problem for a given N, returning a list of queens as Q
nqueens(N, Q) :-
    nqueens(N, N, [], Q).

% pick queens one at a time and test against all previous queens
nqueens(X, N, Qi, Qo) :-
    X > 0,
    pickqueen(X, Y, N),
    not attack(X, Y, Qi),
    X1 is X - 1,
    nqueens(X1, N, [q(X, Y) | Qi], Qo).
nqueens(0, , Q, Q).

% pick a queen for row X.
pickqueen(X, Y, Y) :-
    Y > 0,
    q(X, Y).
pickqueen(X, Y, N) :-
    N > 1,
    N1 is N - 1,
    pickqueen(X, Y, N1).

% check if a queen can attack any previously selected queen
attack(X, _, [q(X, _) | _]). % same row
attack(_, Y, [q(_, Y) | _]). % same col
attack(X, Y, [q(X2, Y2) | _]) :- % same diagonal
    Xd is X2 - X, abs(Xd, Xd2),
    Yd is Y2 - Y, abs(Yd, Yd2),
    Xd2 = Yd2.
attack(X, Y, [_ | T]) :-
    attack(X, Y, T).

q(X, Y) :- not negq(X, Y).
negq(X, Y) :- not q(X, Y).

abs(X, X) :- X >= 0.
abs(X, Y) :- X < 0, Y is X * -1.
```

Figure 7.1: N Queens Program with Lists.

When executed by our prototype implementation the user will get the following:

```
?- nqueens(5,X).
{ nqueens(5,[q(1,2),q(2,4),q(3,1),q(4,3),q(5,5)]), q(1,2), q(2,4),
q(3,1), q(4,3), q(5,5) }
X = [q(1,2),q(2,4),q(3,1),q(4,3),q(5,5)].

?- nqueens(4,X).
{ nqueens(4,[q(1,2),q(2,4),q(3,1),q(4,3)]), q(1,2), q(2,4), q(3,1),
q(4,3) }
X = [q(1,2),q(2,4),q(3,1),q(4,3)];
{ nqueens(4,[q(1,3),q(2,1),q(3,4),q(4,2)]), q(1,3), q(2,1), q(3,4),
q(4,2) }
X = [q(1,3),q(2,1),q(3,4),q(4,2)];
false.
```

As no grounding is performed, multiple instances of the problem can be queried in a single session. The sample output illustrates this by querying both four and five queens. While the entire partial stable model is provided, variables in the query are printed, making desired information much easier to find. In the above example, `X` will be bound to the list of queens selected. As with Prolog interpreters, ';' and '.' can be used to reject or accept a solution, respectively. As there are only two solutions for four queens, pressing ';' a second time leads to failure. Note that the output will often contain variables with names consisting of "Var" followed by an integer. This is simply because s(ASP) renames variables to ensure that they are unique.

```
reachable(V) :- chosen(U, V), reachable(U).
reachable(0) :- chosen(V, 0).

% Every vertex must be reachable.
:- vertex(U), not reachable(U).

% Choose exactly one edge from each vertex.
other(U, V) :-
    vertex(U), vertex(V), vertex(W),
    V \= W, chosen(U, W).
chosen(U, V) :-
    vertex(U), vertex(V),
    edge(U, V), not other(U, V).

% Two edges cannot be incident on the same
% vertex.
:- chosen(U, W), chosen(V, W), U \= V.

% Sample graph: vertexes and the edges connecting them.
vertex(0).
vertex(1).
vertex(2).
vertex(3).
vertex(4).

edge(0, 1).
edge(1, 2).
edge(2, 3).
edge(3, 4).
edge(4, 0).
edge(4, 1).
edge(4, 2).
edge(4, 3).
```

Figure 7.2: A program for Hamiltonian cycle detection with a simple graph included.

7.3.2 Example: Hamiltonian Cycle Detection

Figure 7.2 contains an encoding of the Hamiltonian cycle problem, along with a simple graph. The results of this example provide an interesting look at our use of negatively constrained variables in output. While another solution exists, due to space limitations, only the first is provided. The cycle is represented by the `chosen/2` elements at the beginning of the set:

```
?- reachable(0).
{ chosen(0,1), chosen(1,2), chosen(2,3), chosen(3,4), chosen(4,0),
edge(0,1), edge(1,2), edge(2,3), edge(3,4), edge(4,0), edge(4,1),
edge(4,2), edge(4,3), other(0,0), other(0,2), other(0,3),
other(0,4), other(1,0), other(1,1), other(1,3), other(1,4),
other(2,0), other(2,1), other(2,2), other(2,4), other(3,0),
other(3,1), other(3,2), other(3,3), other(4,1), other(4,2),
other(4,3), other(4,4), reachable(0), reachable(1), reachable(2),
reachable(3), reachable(4), vertex(0), vertex(1), vertex(2),
vertex(3), vertex(4), not chosen(0,0), not chosen(0,2), not
chosen(0,3), not chosen(0,4), not chosen(0,Var644) ( Var644 \= 0,
Var644 \= 1, Var644 \= 2, Var644 \= 3, Var644 \= 4 ), not
chosen(1,0), not chosen(1,1), not chosen(1,3), not chosen(1,4), not
chosen(1,Var710) ( Var710 \= 0, Var710 \= 1, Var710 \= 2, Var710 \=
3, Var710 \= 4 ), not chosen(2,0), not chosen(2,1), not chosen(2,2),
not chosen(2,4), not chosen(2,Var776) ( Var776 \= 0, Var776 \= 1,
Var776\= 2, Var776 \= 3, Var776 \= 4 ), not chosen(3,0), not
chosen(3,1), not chosen(3,2), not chosen(3,3), not chosen(3,Var842)
( Var842 \= 0, Var842 \= 1, Var842 \= 2, Var842 \= 3, Var842 \= 4 ),
not chosen(4,1), not chosen(4,2), not chosen(4,3), not chosen(4,4),
```

```
not chosen(4,Var908) ( Var908 \= 0, Var908 \= 1, Var908 \= 2, Var908
\= 3, Var908 \= 4 ), not chosen(Var627,_) ( Var627 \= 0, Var627 \=
1, Var627 \= 2, Var627 \= 3, Var627 \= 4 ), not chosen(Var663,1) (
Var663 \= 0, Var663 \= 1, Var663 \= 2, Var663 \= 3, Var663 \= 4 ),
not chosen(Var734,2) ( Var734 \= 0, Var734 \= 1, Var734 \= 2, Var734
\= 3, Var734 \= 4 ), not chosen(Var805,3) ( Var805 \= 0, Var805 \=
1, Var805 \= 2, Var805 \= 3, Var805 \= 4 ), not chosen(Var876,4) (
Var876 \= 0, Var876 \= 1, Var876 \= 2, Var876 \= 3, Var876 \= 4 ),
not chosen(Var922,0) ( Var922 \= 0, Var922 \= 1, Var922 \= 2, Var922
\= 3, Var922 \= 4 ), not edge(0,0), not edge(0,2), not edge(0,3),
not edge(0,4), not edge(1,0), not edge(1,1), not edge(1,3), not
edge(1,4), not edge(2,0), not edge(2,1), not edge(2,2), not
edge(2,4), not edge(3,0), not edge(3,1), not edge(3,2), not
edge(3,3), not edge(4,4), not other(0,1), not other(1,2), not
other(2,3), not other(3,4), not other(4,0), not vertex(Var31) (
Var31 \= 0, Var31 \= 1, Var31 \= 2, Var31 \= 3, Var31 \= 4 ) } .
```

7.4 Applications

The s(ASP) system is publicly available (Marple, 2015), and has been used to develop a number of non-trivial applications based on ASP; it has also been used to organize an AI hackathon (UT Dallas AI Society, 2016). Some of these applications cannot be executed on traditional ASP systems such as CLASP, as these applications make use of lists and structures to represent information. They have been developed by people who are not experts in ASP. These applications include:

1. **Degree Audit System:** A system for automatically performing a degree audit of a student's undergraduate transcript at a US University, i.e. , automatically determining

whether a student can graduate with a degree or not, has been developed using the s(ASP) system (Sobhi and Srirangapalli, 2016). The system represents the graduation requirements laid out in the course catalog as ASP clauses. Use of negation is important for representing these requirements. The system has to make use of lists, and has hundreds of courses that appear as constants in the program (hence its grounding will produce an inordinately large program).

2. **Physician Advisory System:** A system for disease management, particularly, for chronic heart failure has been developed using the s(ASP) system (Chen et al., 2016). This system automates the 80-page guidelines (that the American College of Cardiology has developed) by representing them in ASP. While the current system can be run under systems such as CLASP due to the number of constants not being too large, the final system that models a doctor's full knowledge will have quite a few constants, and advanced data-structures may be needed.

3. **Automating Textbook Knowledge:** A system that represents high-school level knowledge about cells (in the discipline of biology) as answer set programs has been developed using s(ASP). It can answer high-school level questions posed as s(ASP) queries. The goal is to represent the knowledge in the entire introductory biology textbook as an answer set program, and then be able to automatically answer questions that would be asked of a student (the questions have to be translated into ASP queries that are then executed to find the answer).

4. **Birthday Gift Advisor:** A recommendation system for birthday gifts has also been developed using the s(ASP) system. This system codes a person's knowledge about friends, level of friendship, a person's wealth level, generosity level, and hobbies as answer set programs. When queried, the system can recommend a birthday present

for a particular friend (e.g., on one's Facebook page). Note that other similar recommendation systems can also be built using s(ASP).

CHAPTER 8

CONCLUSION

8.1 Related Work

Chapter 3 presented a generalization of completion semantics. As mentioned in the introduction, Jürgen Dix has explored various semantics and the properties they share. In his papers, (Dix, 1995a) and (Dix, 1995b), he presents these properties.

The completion semantics covered in this document is just one category of semantics that Dix covers. The main purpose of Dix's work is to identify properties such that "Any semantics satisfying certain properties is uniquely determined by these"(Dix, 1995a). When designing a semantics, these properties can be used to check if the semantics has the desirable behaviour.

Dix's work can be considered foundational in understanding the behaviour of semantics, and according to Google Scholar, at the time of this writing (Dix, 1995a) has 163 citations and (Dix, 1995b) has 201. In fact, a goal-directed algorithm for stable models had been thought to be impossible because it violates the relevance property (Dix, 1995b; Baral and Gelfond, 1994). However, by modifying the program (and query) in such the way that the original answer is easily extracted, dependence on the relevance property was eliminated and such a system was developed, as presented in Section 2.7. However, Dix's work does not provide a generalized way of computing semantics. The work presented in Chapters 3, 4, and 5, on the other hand, while restricted to completion semantics, does this.

More recently, Yanhong A. Liu and Scott D. Stoller (Liu and Stoller, 2016) 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 meta-logical 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 this work.

However, with some modifications to the algorithm and assumptions, such as allowing meta-logical properties and modifying $\mathbf{T}'_{\mathbf{p}}$ to use dual rules only for *complete* (a meta-logical property) predicates instead of all predicates, the generalization presented may be made to compute models for founded and constraint semantics.

Chapter 6 introduced the s(ASP) algorithm. Some of the problems s(ASP) tries to solve has also been explored by others. However, most focus on answer set programming rather than purely stable models. There were several attempts made to develop a goal-directed execution method for propositional answer set programs (Alferes et al., 2004; Bonatti et al., 2008; Bonatti, 2001; Pereira and Pinto, 2005, 2009; Shen et al., 2004). These methods, however, either alter the semantics or significantly restrict the programs accepted. s(ASP) will accept any normal logic program (though it does currently restrict the execution at run-time) without altering the underlying stable model semantics.

Similarly, there has been work in the area of predicate answer set programming, but only with much more severe restrictions on accepted programs (Bonatti, 2004; Heymans and Vermeir, 2003). There has also been research into reducing the limitations of ground program. These ASP systems ground "on the fly", but grounding is still performed at some point during execution (Dal Palù et al., 2009; Dao-Tran et al., 2012; Lefvre and Nicolas, 2009a,b). A variety of efforts have focused on constructive negation (Chan, 1988; Stuckey, 1991; Pearce and Valverde, 2005), but our method's combination of negatively constrained variables and specially adapted dual rules represents a unique approach.

8.2 Further Work

The generalization of completion semantics demonstrated the role of induction and coinduction. This could help with the development of new semantics for specialized cases. It

may be interesting to make use of the knowledge learned with s(ASP) to create a generalized non-ground algorithm. But the majority of future work would propbaly be focused on s(ASP).

s(ASP) itself is new, and with all new things there is a lot to iron out. There are several topic that can be explored. For more efficient execution, a WAM-style abstract machine can be developed for s(ASP). The abstract machine would be able to implement some core features efficiently in a fast language, theoretically giving run-times close to prolog. Other forms of optimizations include better dual rules that do not cause as much backtracking (or at least deals with it more intelligently). At the moment calling a negation has potential to be slow because the backtracking cannot be handled by the programmer since the code is automatically generated. In a similar branch, negation of predicates with a large number of facts have show to be very slow in practice. Much research needs to go into it to figure out why this is happening and how to circumvent it.

The galliwasp system has a feature called dcc (Marple and Gupta, 2014). This reorders NMR checks to more efficiently compute the partial models, and has the additional benefit of computing a reasonable answer when the program is globally consistent. Doing for the predicate case is much harder.

In some cases, the fact that we have consistency checks for non-OLON rules may cause the system to do more work than necessary. Research must be done to identify if this is a problem and what solution there are, if it is.

There is also the topic of disunifying unbound variables. A possible solution is a specialized constraint solver. Is this feasible? What will it require? Are there other work arounds? There are many questions that need to be answered.

Finally, s(ASP) is just a bare-bones algorithm. There are many options to explore for extending it. This includes, but is not limited to, porting extensions/libraries from prolog and implementing a library of built-in predicates for users to use.

8.3 Conclusion

Chapters 3, 4, and 5 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.

Chapter 3 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.

Chapter 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 Chapter 5.

Chapters 6 and 7 presented a method for computing partial stable models of predicate normal logic programs and proven it sound for a large class of programs. The key to this method is the use of a special, non-Herbrand universe which allows us to ensure soundness while still producing useful results for a large class of programs. The method also relies on coinduction and constructive negation to execute programs in a top-down manner similar to that used in Prolog systems. Compared to similar attempts, this method supports a much larger class of programs. Only two restrictions are placed on the programs execution: that arithmetic operations must be ground when executed and that two negatively constrained variables cannot be disunified with each other. An implementation of our method, s(ASP), is freely available (Marple, 2015) and has already been used in the development of non-trivial applications.

REFERENCES

Alferes, J. J., L. M. Pereira, and T. Swift (2004, July). Abduction in Well-Founded Semantics and Generalized Stable Models via Tabled Dual Programs. *Theory and Practice of Logic Programming 4*, 383–428.

Apt, K. R. and H. A. Blair (1990). Arithmetic Classification of Perfect Models of Stratified Programs. *Fundamenta Informaticae 13*(1), 1–17.

Apt, K. R. and R. N. Bol (1994). Logic programming and negation: A survey. *J. Log. Program. 19/20*, 9–71.

Bansal, A., N. Saeedloei, and G. Gupta (2010). Timed Planning. In *Proceedings of the Twenty-Third International Florida Artificial Intelligence Research Society Conference*. AAAI Press.

Baral, C. (2003). *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

Baral, C. and M. Gelfond (1994). Logic Programming and Knowledge Representation. *The Journal of Logic Programming 19*, 73–148.

Bonatti, P. A. (2001, November). Resolution for Skeptical Stable Model Semantics. *Journal of Automated Reasoning 27*, 391–421.

Bonatti, P. A. (2004). Reasoning with Infinite Stable Models. *Artificial Intelligence 156*(1), 75–111.

Bonatti, P. A., E. Pontelli, and T. C. Son (2008). Credulous Resolution for Answer Set Programming. In *Proceedings of the 23rd national conference on Artificial Intelligence - Volume 1*, AAAI'08, pp. 418–423. AAAI Press.

Cadoli, M. and M. Schaerf (1993). A Survey of Complexity Results for Non-Monotonic Logics. *The Journal of Logic Programming 17*(2-4), 127–160.

Chan, D. (1988). Constructive Negation Based on the Completed Database. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, pp. 111–125. MIT Press.

Chen, W., T. Swift, and D. S. Warren (1995). Efficient Top-Down Computation of Queries Under the Well-Founded Semantics. *J. Log. Program. 24*(3), 161–199.

Chen, Z., K. Marple, E. Salazar, G. Gupta, and L. Tamil (2016). A Physician Advisory System for Chronic Heart Failure Management Based on Knowledge Patterns. Submitted to ICLP'16.

Dal Palù, A., A. Dovier, E. Pontelli, and G. Rossi (2009). GASP: Answer Set Programming with Lazy Grounding. *Fundamenta Informaticae 96*(3), 297–322.

Dao-Tran, M., T. Eiter, M. Fink, G. Weidinger, and A. Weinzierl (2012). OMiGA: An Open Minded Grounding On-The-Fly Answer Set Solver. In *Logics in Artificial Intelligence*, Volume 7519 of *Lecture Notes in Computer Science*, pp. 480–483. Springer Berlin Heidelberg.

Dix, J. (1995a). A classification theory of semantics of normal logic programs: I. strong properties.

Dix, J. (1995b). A classification theory of semantics of normal logic programs: Ii. weak properties.

Fitting, M. and M. 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)*, pp. 1054–1069.

Gebser, M., B. Kaufmann, A. Neumann, and T. Schaub (2007). Clasp: A Conflict-Driven Answer Set Solver. In *Proceedings of the 9th international conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR'07, pp. 260–265. Springer-Verlag.

Gelfond, M. and V. 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)*, pp. 1070–1080.

Gupta, G., A. Bansal, R. Min, L. Simon, and A. Mallya (2007). Coinductive Logic Programming and Its Applications. In *Proceedings of the 23rd international conference on Logic Programming*, ICLP'07, pp. 27–44. Springer-Verlag.

Gupta, G., K. Marple, B. DeVries, F. Kluniak, R. Min, N. Saeedloei, and T. Haage (2012). Coinductive answer set programming or consistency-based computing. Co-LP 2012 - A workshop on Coinductive Logic Programming.

Gupta, G., N. Saeedloei, B. DeVries, R. Min, K. Marple, and F. Kluźniak (2011). Infinite computation, co-induction and computational logic. In A. Corradini, B. Klin, and C. Cîrstea (Eds.), *Algebra and Coalgebra in Computer Science*, Berlin, Heidelberg, pp. 40–54. Springer Berlin Heidelberg.

Heymans, S. and D. Vermeir (2003). Integrating Semantic Web Reasoning and Answer Set Programming. In *Proceedings of the 2nd International ASP Workshop*, ASP'03, pp. 194–208. CEUR-WS.org.

Jacobs, B. and J. Rutten (1997). A tutorial on (co)algebras and (co)induction. *EATCS Bulletin 62*, 62–222.

Jaffar, J. and J.-L. Lassez (1987). Constraint Logic Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL'87, pp. 111–119. ACM.

Lefvre, C. and P. Nicolas (2009a). A First Order Forward Chaining Approach for Answer Set Computing. In E. Erdem, F. Lin, and T. Schaub (Eds.), *Logic Programming and Nonmonotonic Reasoning*, Volume 5753 of *Lecture Notes in Computer Science*, pp. 196–208. Springer Berlin Heidelberg.

Lefvre, C. and P. Nicolas (2009b). The First Version of a New ASP Solver: ASPeRiX. In E. Erdem, F. Lin, and T. Schaub (Eds.), *Logic Programming and Nonmonotonic Reasoning*, Volume 5753 of *Lecture Notes in Computer Science*, pp. 522–527. Springer Berlin Heidelberg.

Leinster, T. (2016). Basic category theory.

Lifschitz, V. and H. Turner (1994). Splitting a Logic Program. In *Proceedings of the Eleventh International Conference on Logic Programming*, ICLP'94, pp. 23–37. MIT Press.

Lin, F. and Y. Zhao (2004). ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. *Artif. Intell. 157*(1-2), 115–137.

Liu, Y. A. and S. D. Stoller (2016). The founded semantics and constraint semantics of logic rules. *CoRR abs/1606.06269*.

Lloyd, J. (1987). *Foundations of Logic Programming*. Symbolic Computation: Artificial Intelligence. Springer-Verlag.

Marple, K. (2014a). *Design and Implementation of a Goal-directed Answer Set Programming System*. Ph. D. thesis, University of Texas at Dallas.

Marple, K. (2014b). Galliwasp. `http://galliwasp.sourceforge.net`.

Marple, K. (2015). s(ASP). `https://sourceforge.net/projects/sasp-system/`.

Marple, K., A. Bansal, R. Min, and G. Gupta (2012a). Goal-directed Execution of Answer Set Programs. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, PPDP '12, New York, NY, USA, pp. 35–44. ACM.

Marple, K., A. Bansal, R. Min, and G. Gupta (2012b). Goal-directed execution of answer set programs. In *Principles and Practice of Declarative Programming, PPDP'12, Leuven, Belgium - September 19 - 21, 2012*, pp. 35–44.

Marple, K. and G. Gupta (2013). Galliwasp: A Goal-Directed Answer Set Solver. In *Logic-Based Program Synthesis and Transformation*, Volume 7844 of *Lecture Notes in Computer Science*, pp. 122–136. Springer Berlin Heidelberg.

Marple, K. and G. Gupta (2014). Dynamic Consistency Checking in Goal-Directed Answer Set Programming. *Theory and Practice of Logic Programming 14*(4–5), 415–427.

Min, R. K. (2009). *Predicate Answer Set Programming with Coinduction*. Ph. D. thesis, Richardson, TX, USA. AAI3375960.

Minker, J. (Ed.) (1988). *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann Publishers Inc.

Pearce, D. and A. Valverde (2005). A Order Nonmonotonic Extension of Constructive Logic. *Studia Logica 80*(2–3), 321–346.

Pereira, L. and A. Pinto (2005). Revised Stable Models - A Semantics for Logic Programs. In *Progress in Artificial Intelligence*, Volume 3808 of *Lecture Notes in Computer Science*, pp. 29–42. Springer-Verlag.

Pereira, L. and A. Pinto (2009). Layered Models Top-Down Querying of Normal Logic Programs. In *Practical Aspects of Declarative Languages*, Volume 5418 of *Lecture Notes in Computer Science*, pp. 254–268. Springer-Verlag.

Roşu, G. and D. Lucanu (2009). Circular coinduction: A proof theoretical foundation. In A. Kurz, M. Lenisa, and A. Tarlecki (Eds.), *Algebra and Coalgebra in Computer Science*, Berlin, Heidelberg, pp. 127–144. Springer Berlin Heidelberg.

Schlipf, J. S. (1990). The Expressive Powers of the Logic Programming Semantics. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 196–204. ACM.

Shen, Y., J. You, and L. Yuan (2004). Enhancing Global SLS-Resolution with Loop Cutting and Tabling Mechanisms. *Theoretical Computer Science 328*(3), 271–287.

Simon, L., A. Bansal, A. Mallya, and G. 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*, pp. 472–483.

Sobhi, A. and S. Srirangapalli (2016). Graduation Audit System. `https://gitlab.com/saikiran1096/gradaudit/`.

Sterling, L. and E. Y. Shapiro (1994). *The Art of Prolog - Advanced Programming Techniques, 2nd Ed.* MIT Press.

Stuckey, P. (1991, July). Constructive Negation for Constraint Logic Programming. In *Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science*, LICS'91, pp. 328–339. IEEE Computer Society.

Swift, T. S. and D. S. Warren (2012). XSB: Extending Prolog with Tabled Logic Programming. *TPLP 12*(1-2), 157–187.

Ullman, J. (1994). Assigning an Appropriate Meaning to Database Logic with Negation. In *Computers as Our Better Partners*, pp. 216–225. World Scientific Press.

UT Dallas AI Society (2016). s(ASP) Hackathon. `https://hackai16.devpost.com/submissions`.

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

BIOGRAPHICAL SKETCH

Elmer Salazar was born and raised in the Californian central valley. There he recieved his Bachelor of Science in Computer Science and Physics in December 2005 from The California State University in Stanilaus. After graduating, he took a job teaching 6th to 12th grade technology in junior high and high school. After three years, he decided to go back to school, gaining a Master of Science in Computer Science at The University of Texas at Dallas, and then continuing for a PhD.

CURRICULUM VITAE

RESEARCH INTERESTS

Nonmonotonic Computational Logic, Artificial Intellegence, Machine Learning, Common Sense Reasoning

EDUCATION

Bachelor of Science, Computer Science and Physics
California State University in Stanislaus, Turlock, CA, December 2005

Masters of Science, Computer Science
Univerity of Texas in Dallas, Richardson, TX, December 2013

Ph.D., Computer Science
Univerity of Texas in Dallas, Richardson, TX, expected August 2019

EXPERIENCE

Tutor (Workshop Facilitator) 2001-2003
California State University in Stanislaus, Turlock, CA

- Biweekly workshops where students can get help understanding concepts from "foundational" lower level courses.

6^{th}-12^{th} Technology Teacher Aug. 2006 - May 2009
Gustine ISD, Gustine TX

- Full time teaching position for 6th - 12th grade computer courses
- Used Self-Developed Curriculum

Teacher's Assistant 2016 - Current
University of Texas at Dallas, Richardson, TX

- Duties included grading and helping students understand the material
- Always got good reviews from my professors.

Research Assistant 2016 - Current
University of Texas at Dallas, Richardson, TX

- Studing knowledge representation for asp systems

PUBLICATIONS

Arias, J., M. Carro, E. Salazar, K. Marple, and G. Gupta (2018). Constraint answer set programming without grounding. *TPLP 18*(3-4), 337–354.

Chen, Z., K. Marple, E. Salazar, G. Gupta, and L. Tamil (2016). A physician advisory system for chronic heart failure management based on knowledge patterns. *TPLP 16*(5-6), 604–618.

Chen, Z., E. Salazar, K. Marple, G. Gupta, L. Tamil, D. Cheeran, S. Das, and A. Amin (2017). Improving adherence to heart failure management guidelines via abductive reasoning. *TPLP 17*(5-6), 764–779.

Chen, Z., E. Salazar, K. Marple, G. Gupta, L. Tamil, S. Das, and A. Amin (2017). Improving adherence to heart failure management guidelines via abductive reasoning. *CoRR abs/1707.04957*.

Chen, Z., E. Salazar, K. Marple, S. R. Das, A. Amin, D. Cheeran, L. Tamil, and G. Gupta (2018, 11). An ai-based heart failure treatment adviser system. *IEEE Journal of Translational Engineering in Health and Medicine PP*, 1–1.

Gupta, G., E. Salazar, K. Marple, Z. Chen, and F. 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, Gothenburg, Sweden, 6th August 2017*, pp. 64–68.

Shakerin, F., E. Salazar, and G. Gupta (2017). A new algorithm to automate inductive learning of default theories. *TPLP 17*(5-6), 1010–1026.

PLANNED WORK

Interaction Between Probabilistic and Logic Models to Answer Richer Questions
From observation, it seems models of data based on probabilities that are often learned through techniques such as neural networks and SVMs and logic-based models complement each other. This topic is an exploration of this idea. The idea is to use both model types of the same data to answer richer questions implied by the data.
Hierarchical Inductive Logic: Learning More Meaningful Logic Programs
Current learning algorithms based on logic seek high accuracy, but does not try to make the resulting logic program meaningful to humans. HIL is an attempt to do just that by mimicking the way programmers write programs: breaking the problem into smaller meaningful subproblems.
Learning With Justification
In a similar vein as HIL, this topic explores the idea of having each example justified by a

human. The idea is the human's thought process would be captured in the justification and used to guide the hypothesis search.

Dynamic Consistency Check for s(ASP)

The current s(ASP) system is young and there are problems that stop it from being useful as a generic tool. One such problem is the excessive backtracking that can be caused by the nmr consistency checks. DCC is a technique of reordering checks to ensure early failure. However, it is not so straight forward when the program must be executed unground.

Witness Instantiation

Another problem limiting the usage of s(ASP) is the fact that variables cannot be constrained against each other without an inefficient constraint solver specifically designed for this case. Witness instantiation is a technique still in the early idea phase where we treat existential variables at runtime as if they are a ground constant. There is still much that must be worked out, but if successful it will lead to a more efficient treatment of disunification constraints.